DOCUMENT RESUME

ED 143 508 SE 022 985

AUTHOR Charp, Sylvia; And Others
TITLE Algorithms, Computation and Mathematics (Algol Supplement). Student Text. Revised Edition.
INSTITUTION Stanford Univ., Calif. School Mathematics Study Group.
SPONS AGENCY National Science Foundation, Washington, D.C.
PUB DATE 66
NOTE 138p.; For related documents, see SE 022 983-988; Not available in hard copy due to marginal legibility of original document

EDRS PRICE MF-$0.83 Plus Postage. HC Not Available from EDRS.
DESCRIPTORS Algorithms; *Computers; *Instructional Materials; *Programing Languages; Secondary Education; *Secondary School Mathematics; *Textbooks
IDENTIFIERS *ALGOL; *School Mathematics Study Group

ABSTRACT

This is the student's textbook for Algorithms, Computation, and Mathematics (Algol Supplement). This computer language supplement is split off from the main text to enable a school to choose the computer language desired, and also to make it easier to modify the course as languages change. The chapters in the text are designed to add language capability. Each can be read in conjunction with the main text section by section. (RH)

****
* Documents acquired by ERIC include many informal unpublished *
* materials not available from other sources. ERIC makes every effort *
* to obtain the best copy available. Nevertheless, items of marginal *
* reproducibility are often encountered and this affects the quality *
* of the microfiche and hardcopy reproductions ERIC makes available *
* via the ERIC Document Reproduction Service (EDRS). EDRS is not *
* responsible for the quality of the original document. Reproductions *
* supplied by EDRS are the best that can be made from the original. *
****

ED143508

U S DEPARTMENT OF HEALTH, EDUCATION & WELFARE NATIONAL INSTITUTE OF EDUCATION

THIS DOCUMENT HAS BEEN REPRODUCED EXACTLY AS RECEIVED FROM THE PERSON OR ORGANIZATION ORIGINATING IT. POINTS OF VIEW OR OPINIONS STATED DO NOT NECESSARILY REPRESENT OFFICIAL NATIONAL INSTITUTE OF EDUCATION POSITION OR POLICY

"PERMISSION TO REPRODUCE THIS MATERIAL HAS BEEN GRANTED BY

SMSG

TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC) AND THE ERIC SYSTEM CONTRACTORS."

ALGORITHMS, COMPUTATION AND MATHEMATICS

(Algol Supplement)

Student Text

Revised Edition

The following is a list of all those who participated in the preparation of this volume:

Sylvia Charp, Dobbins Technical High School, Philadelphia, Pennsylvania
Alexandra Forsythe, Gunn High School, Palo Alto, California
Bernard A. Galler, University of Michigan, Ann Arbor, Michigan
John G. Herriot, Stanford University, California
Walter Hoffmann, Wayne State University, Detroit, Michigan
Thomas E. Hull, University of Toronto, Toronto, Ontario, Canada
Thomas A. Keenan, University of Rochester, Rochester, New York
Robert E. Monroe, Wayne State University, Detroit, Michigan
Silvio O. Navarro, University of Kentucky, Lexington, Kentucky
Elliott I. Organick, University of Houston, Houston, Texas
Jesse Peckenham, Oakland Unified School District, Oakland, California
George A. Robinson, Argonne National Laboratory, Argonne, Illinois
Phillip M. Sherman, Bell Telephone Laboratories, Murray Hill, New Jersey
Robert E. Smith, Control Data Corporation, St. Paul, Minnesota
Warren Stenberg, University of Minnesota, Minneapolis, Minnesota
Harley Tillitt, U. S. Naval Ordnance Test Station, China Lake, California
Lyneve Waldrop, Newton South High School, Newton, Massachusetts

The following were the principal consultants:

George E. Forsythe, Stanford University, California
Bernard A. Galler, University of Michigan, Ann Arbor, Michigan
Wallace Givens, Argonne National Laboratory, Argonne, Illinois

SE 022 985

© 1965 and 1966 by The Board of Trustees of the Leland Stanford Junior University
All rights reserved
Printed in the United States of America

Permission to make verbatim use of material in this book must be secured from the Director of SMSG. Such permission will be granted except in unusual circumstances. Publications incorporating SMSG materials must include both an acknowledgment of the SMSG copyright (Yale University or Stanford University, as the case may be) and a disclaimer of SMSG endorsement. Exclusive license will not be granted save in exceptional circumstances, and then only by specific action of the Advisory Board of SMSG.

Financial support for the School Mathematics Study Group has been provided by the National Science Foundation.

TABLE OF CONTENTS

Chapter

A2 ALGOL LANGUAGE MANUAL TO ACCOMPANY CHAPTER 2

A2-1. Introduction 1
A2-2. ALGOL language elements. 7
A2-3. Input-output statements. 15
A2-4. Assignment statements. 22
A2-5. The order of computation in an ALGOL expression. . . . 31
A2-6. Meaning of assignment when the variable on the left is of different type from the expression on the right. 32
A2-7. Writing complete ALGOL programs. 35
A2-8. Alphanumeric data. 39

A3 BRANCHING AND SUBSCRIPTED VARIABLES

A3-1. Conditional statements 45
A3-2. Auxiliary variables. 60
A3-3. Compound condition boxes and multiple branching. . . . 62
A3-4. Precedence levels for relations. 72
A3-5. Subscripted variables. 73
A3-6. Double subscripts. 78

A4 LOOPING

A4-1. The "for clause" and the "for statement" 81
A4-2. Illustrative examples. 87
A4-3. Table-look-up. 91
A4-4. Nested loops 93

A5 PROCEDURES

A5-1. Procedures 97
A5-2. Functions and ALGOL. 102
A5-3. ALGOL function procedures. 103
A5-4. ALGOL "proper" procedures. 106
A5-5. Alternate exits and techniques for branching 109
A5-6. Symbol manipulation in ALGOL 111

A7 SOME MATHEMATICAL APPLICATIONS

A7-1. Root of an equation by bisection 119
A7-2. The area under a curve: an example, $y = 1/x$, between $x = 1$ and $x = 2$. . 126
A7-3. Area under curve: the general case. 128
A7-4. Simultaneous linear equations: Developing a systematic method of solution 130
A7-5. Simultaneous linear equations: Gauss algorithm. . . . 131

Chapter A2

ALGOL LANGUAGE MANUAL TO ACCOMPANY CHAPTER 2

A2-1 Introduction

In Chapter 2 we developed an appreciation of input, output and assignment steps as components of algorithms expressed in the form of flow charts. So far, we have viewed flow charts as a means for conveying a sequence of computation rules primarily from one person to another. We have tacitly assumed that only man can read, understand and carry out the intent of such flow charts. Naturally we want to include computers in the set of all things which can read, understand and carry out procedures.

ALGOL--language and processor

Programming languages like ALGOL and FORTRAN accomplish this objective. The steps of a programming language are called statements. They correspond roughly to the boxes of a flow chart.

ALGOL 60

Several years ago a group of computer specialists and mathematicians from many countries jointly developed an English-like programming language which they called ALGOL 60. The language was designed with these objectives.

1. A wide variety of algorithms can be described with this language. Its chief area of application is for expressing algorithms which deal with scientific and engineering computation. Algorithms for many other types of problems can also be expressed satisfactorily in ALGOL 60.

2. The rules or "grammar" of the language are defined precisely--i.e., with mathematical rigor (unlike English). The net result is that an algorithm written in ALGOL 60 means precisely the same thing to each person who reads it, i.e., it means the same thing all over the world to people who have learned this language. Hence, ALGOL 60 provides

a means for communicating algorithms via correspondence and publications, from one person to another. It is indeed an international language.

3. With minor additions or modifications, ALGOL 60 can be "implemented" on many types of digital computers. By implementation we mean that a computer can be programmed to accept as "source" programs algorithms written in a version of ALGOL 60 and automatically convert them to sequences of computer instructions often called "target" programs, which can then be executed by the computer.

These features of ALGOL 60 have led to its wide, international acceptance among mathematicians and scientists. In this chapter, we shall begin to describe ALGOL 60 as it is typically implemented for use on a computer, calling it simply ALGOL.

Each statement in ALGOL is written so that when typed or punched on a card it can be transferred to the computer's memory. Here it can be scanned character by character† and analyzed for its full intent.

Programs which analyze these statements are called compiler or processor programs. A typical ALGOL processor reads statements originating on punch cards and analyzes them by converting each statement into an equivalent sequence of computer instructions. Were these instructions to be executed, it would amoung to having the computer carry out the intent of the ALGOL statement.

Target programs and source programs

The processor program will read and analyze all the statements of an ALGOL program to generate a complete set of instructions or "target" program, sufficient, if executed, to carry out the intent of the entire process. The "target" program gets its name because it is the target or objective of the processor program. Similarly, the processor has received as input a "source" program written in the ALGOL language. When the processor program finishes generating the target program, the computer may execute these instructions right away. This is feasible because the target program is developed and kept in the computer's memory. If the target program is too big to fit in high speed memory along with the large processor program, the generated target is stored temporarily in some form of auxiliary memory media such as on magnetic drums, disks, or tapes, or in the form of decks of punched cards. When stored in auxiliary

†This process is further described in Section 2-4 of the main text.

memory media, especially punched cards, target programs can be recalled for execution at any subsequent time, as suggested in Figure A2-1. We would rarely wish to read or study the target program ourselves, but in principle this can always be done by causing the processor to print the target program.

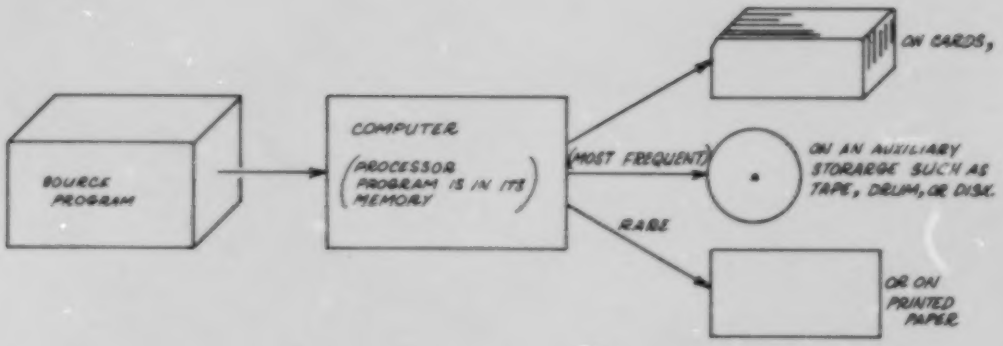


Figure A2-1. The "compiling" process

It may be intriguing to you to learn how the processor program does its job. After all, it is also a flow-chartable process and hence could easily be within our abilities to understand it. Chapter 8 will shed some light on this interesting process. For the present, however, we will avoid any head-on discussion of this topic because our first interest must be to learn to write simple algorithms for solving mathematical problems in ALGOL. We will, however, be making occasional comments that bear indirectly on the nature and structure of the processor.

General appearance of an ALGOL program

Recall the process for computing

$$D = \sqrt{A^2 + B^2 + C^2}$$

whose flow chart we displayed in Figure 2-4. Each box can be written as an ALGOL statement, as shown in Figure A2-2.

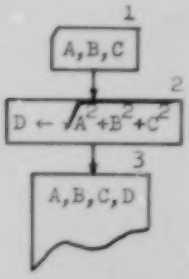
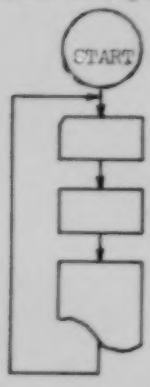
	Flow chart form	ALGOL statement
input		read(A,B,C);
assignment		D := sqrt(A↑2 + B↑2 + C↑2);
output		write(A,B,C,D);

Figure A2-2. ALGOL statements compared with flow chart boxes

Notice the similarity between the boxes and the ALGOL statements. The differences are largely superficial; that is, the symbols used in each case may be different, but the ideas appear to be the same.

We don't have to connect the statements with arrows, because, when we write ALGOL statements one below the other, we imply that they are to be carried out one after the other from top to bottom. In order to suggest repeating the process for many sets of data, we drew a line from box 3 back to box 1 in the flow chart. There is an analogy in ALGOL to accomplish the same objective. We simply give the "read" statement a name or label, say START. Then we add after the "write" a statement which "sends control" to the designated statement. This is shown in Figure A2-3.




```
START:  read(A,B,C);

        D := sqrt(A↑2 + B↑2 + C↑2);

        write(A,B,C,D);

        go to START;
```

Figure A2-3. The go to statement directs control to any desired statement

In other words, we give the "read" statement the label START and then introduce an ALGOL statement,

go to START;

for the purpose of indicating a jump or transfer to that statement. The jump statement consists of the special symbol "go to" followed by a label. Because it reads like English, the jump statement is easy to understand and we shall say no more about it at this time.

As we focus on a new language, it always takes a while for its features, its special symbols and punctuation patterns to stand out clearly. If you are observant, however, you have probably concluded, and correctly, that in ALGOL:

(1) statements are separated by the semicolon (;).
As a reminder that the semicolon is needed for this purpose, we will usually show the semicolon at the end of the statement even when the statement stands alone.

(2) a statement may have a label. If so, the label precedes the statement and is separated from it by a colon (:).

(3) the assignment symbol is a colon followed by an equal sign (:=).
Alas, a left-pointing arrow would have been our choice!

(4) the symbol for exponentiation is the upward-pointing arrow (↑); and we appear to have lost the ability to use a symbol like $\sqrt{\ }$, having now to use sqrt with the argument following it enclosed in parentheses.

(5) for some curious reason certain words are underlined. We cannot yet guess what, if any, significance to attach to this.

Now, before taking a more methodical look at input, output and assignment statements and rules for forming these correctly, we show in Figure A2-4 a complete ALGOL program and discuss it briefly.

```
begin                comment  Evaluation of D;
                     real A, B, C, D;
               START: read(A,B,C);
                     D := sqrt(A↑2 + B↑2 + C↑2);
                     write (A, B, C, D);
                     go to START;
end
```

Figure A2-4. Complete ALGOL program

Card layout

Under control of an ALGOL processor, the computer will read statement by statement this program which is punched on cards. It is a good idea to place each statement on a separate card or at least to begin each statement on a new card. In this way the program becomes easier to read and proofread. Strictly speaking, however, since the semicolon acts as a separating mark between statements, it is possible to pack more than one statement on a card. If a statement is too long to fit on one card, it may be continued on succeeding cards.

Figure A2-5 shows how our program may be punched on cards to form a "program deck".

You will recall that today's key punches do not punch special cod s for lower case letters. Computer implementations of ALGOL which accept programs input on punched cards are designed to expect only capital lett rs. This explains part of the difference you see between Figures A2-4 and A2-5.

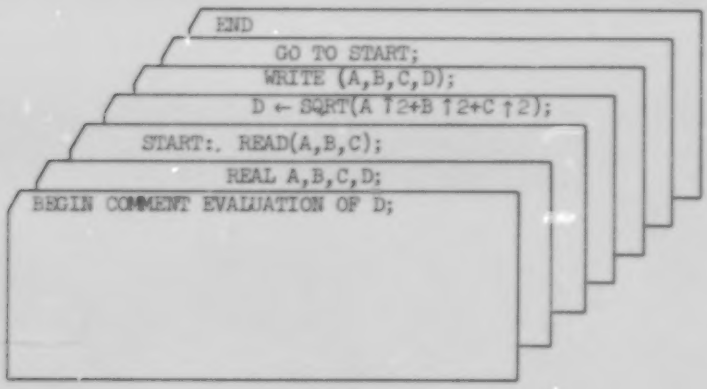


Figure A2-5. The program as a "deck" of cards

The key punch used to prepare these cards is not typical. It has special punches for characters like the : ; ← which are useful for preparing ALGOL programs. In this case the character ← is recognized to mean := .

If you prepare ALGOL programs on standard key punches which have character sets such as that shown in Figure 1-18, other compromises will have to be made. Your instructor will explain these details.

A2-2 ALGOL language elements

We are now ready to begin a detailed look at ALGOL, starting with ALGOL's "official" alphabet, or character set, and the various types of symbols which may be formed from these characters. Among these are the numerals (constants), variables, labels, names of functions, operators, and special symbols.

Language

The program of Figure A2-4 begins and ends with the special (underlined) symbols begin and end. We can almost guess the intent of the first two lines. The special symbol comment is placed there to identify, in a purely descriptive way, the title of the program. Comments help to make the ALGOL self-explanatory. The special symbol real identifies the type of numerical values assigned to the variables listed after real. In this case each of the variables A, B, C, D has a value corresponding to a real number.† We shall refer to a line like

real A, B, C, D;

as a "declaration" to distinguish it from a statement. Declarations describe the variables and other components of the algorithm to the compiler program. They are, in a sense, passive. Statements, on the other hand, are imperatives expressing action. Only statements can correspond to the boxes of a flow chart.

In summary we see that an ALGOL program appears to consist of a group of statements preceded by a group of one or more declarations. The symbols begin and end sandwich the two groups, i.e., they perform the same function as parentheses. An ALGOL program can be punched as a deck of cards, thereby becoming machine readable. To do this requires transliteration into the more restricted character set of the key punch.

---

†Naturally, not all real numbers can be represented accurately inside a computer as there is only a fixed (finite) precision for each number stored in a word of the computer's memory. This precision, in turn, may depend on the computer for which ALGOL 60 has been implemented.

The ALGOL character set

The characters which are used in ALGOL are shown in Table A2-1.

Table A2-1

The ALGOL Character Set

(a) Letters

a	b	c	d	e	f	g	h	i
j	k	l	m	n	o	p	q	r
	s	t	u	v	w	x	y	z
A	B	C	D	E	F	G	H	I
J	K	L	M	N	O	P	Q	R
	S	T	U	V	W	X	Y	Z

(b) Digits

0	1	2	3	4	5	6	7	8	9

(c) Special Characters

+	-	×	/	↑		
.	10	,	:	:=	;	"
(	)	[	]			
=	≠	<	≤	>	≥	

Constructing numerals (numerical constants)

A numeral may have several parts, some of which may be omitted when not essential. These are:

1. Sign
2. The numeral itself which may be written with
   a) a fractional part only
   b) an integral part only
   c) a fractional and an integral part

3. Scale factor to the base 10 (written as an integer with or without a sign)

Listed below are some example numerals. To the right of each numeral are the parts, as defined above, out of which the numeral is constructed. Study these examples carefully.

Examples	Constructed from parts		
- .0059	1, 2a		
+ .59	1, 2a		
.6924	2a		
0	2b		
155	2b		
- 4	1, 2b		
4.0	2c		
- 154.04	1, 2c		
$4_{10}-7$	2b, 3	means	4×10^{-7}
$-.596_{10}+3$	1, 2a, 3	means	$-.596 \times 10^{3}$
10^{16}	3	means	10^{16}
$15.246_{10}1$	2c, 3	means	15.246×10^{1}
$-_{10}-4$	1, 3	means	-10^{-4}

Exercise A2-2 Set A

Identify the components which go into forming these numbers

$-14_{10}+04$

52.0

-017.14

10^{1}

$+10_{10}10$

One type of numeral which we are used to writing is outlawed in ALGOL. This is an integer with a decimal point to its right. Thus

4. is invalid but 4 or 4.0 is valid

-17. is " " -17 or -17.0 is valid

A way to remember that a numeral like 4. is invalid is to imagine that the fractional part of the number, suggested by the presence of the decimal point, is missing.

Constructing variables, labels and names for functions

Variables, labels, and function names are called identifiers in the reference literature of ALGOL. Any of these may be formed according to the same rule: A letter followed by a sequence of letters or digits constitutes an identifier.

Examples

Harry Temp x T46

IKE COLUMN A15AA

In principle there is no restriction to the length of an identifier but from a practical consideration we will restrict them to six or fewer letters and/or digits.†

Exercise A2-2 Set B

Each of the following groups of characters would be invalid as an individual identifier:

2JOHN M/4 T.6 F-6

Explain why.

Variables--integer versus real

We must agree to think of every arithmetic variable of our program as either type real or type integer.

At the top or "head" of every ALGOL program we will state or "declare" the type of each variable used in the program and, in this way, will help the processor program to properly analyze the statements which follow.

To declare the types of our variables, we simply group all those of integer type in one list and all those of real type in another list.

---

†Some ALGOL implementations restrict identifiers to six characters. Others accept more than six characters but keep track of only the leading six characters.

Examples

integer I, P, q, PRES, COUNT ;

real height, X, ORD, y3 ;

The underlining of integer and real will be explained at the end of the section under the heading Special Symbols.

As you know, the set of all real numbers includes the integers, so a variable of type real may have a value which is integral, i.e., one which has no fractional part. Thus, legitimate values for real variables are

-4, 2.1, 16, -0.149, 17.0

On the other hand, a variable of type integer cannot have a value with a fractional part. Thus, the values

.4, -1.1, $17\frac{1}{3}$

obviously cannot be assigned to variables declared to be of type integer.

Names of standard mathematical functions

Certain identifiers are reserved to refer to names of standard mathematical functions as shown in Table A2-2. Notice the special way we spell these functions.

Table A2-2

Standard Mathematical Functions

	ALGOL identifier (name)	meaning
1.	abs	absolute value
2.	sqrt	square root
3.	ln	logarithm to the base e
4.	exp	powers of e, or exponential
5.	sin	sine of an angle whose measure is given in radians
6.	cos	cosine of an angle whose measure is given in radians

Table A2-2 Standard Mathematical Functions, continued

7.	arctan	arctangent, or principal angle in radians of a given tangent value. (That is, arctan(x) gives a value in radians corresponding to the principal angle whose tangent is x).
8.	entier	greatest integer function. (That is, entier(x) means [x].)
9.	sign	sign of a number. (That is, sign(x) means $x/\|x\|$, unless x = 0 in which case sign(x) means 0.)

By now you are probably familiar with all of these mathematical functions. "entier" is the name chosen in ALGOL for the greatest integer function which has been discussed thoroughly in chapter 2. The functions "sin", "cos" and "arctan" will of course be familiar to you if you have studied trigonometry. A use for the "sign" function, perhaps the least familiar of these, was suggested in section 2-5.

When we use any one of these standard functions in an ALGOL statement, the resulting machine code (target program) automatically carries out the evaluation of the specified function.

Operators

To write arithmetic expressions and assignment statements we need symbols for arithmetic operations. The characters chosen for these operator symbols are:

. + - × ↑ ÷ /

In Table A2-3 we show a list of the symbols we shall be using for the various arithmetic operators. For convenience in later discussion they are given in hierarchical order, that is, in descending order of precedence which is the same in ALGOL as it is in our flow chart language. We have also included the assignment symbol := in this group. It is a binary operator, but, of course, not really arithmetic in nature.

↑pronounced on-t-aye

Table A2-3

ALGOL Operator Symbols

	Symbol	Meaning	Example
	↑	Exponentiation	A↑B means A^B
same level of precedence	×	Multiplication	A×B
	/	Division (real)	A/B
	÷	Division (integer)	A÷B means $\text{sign}(\frac{A}{B}) \times \lfloor \lvert \frac{A}{B} \rvert \rfloor$
same level of precedence	+	Addition	A + B
	-	Subtraction	A - B
	:=	Assignment	A := B means A ← B

Special symbols

Certain symbols which at first glance resemble English words appear underlined in ALGOL programs. We have already seen a few examples:

begin end comment real integer go to.

These symbols play a special role in ALGOL programs and are not to be confused with identifiers. They will not be confused with identifiers if they are marked or printed in a distinctive fashion. Texts printed commercially usually use boldface letters or place such symbols in quotes. In this book we underline such symbols.

The following ALGOL statements,

```
        real T;
        T := 0;
real:   T := T + 1;
        go to real;
```

while somewhat confusing at first glance, are perfectly correct here. The word real is used as a statement label and is not at all the same as the special symbol real.

The ALGOL special symbols which were described above as underlined in this book present a problem for punch card input. There is no way to underline on a keypunch. Burroughs ALGOL solves this dilemma by decreeing that special symbols may not be used as identifiers. In other words, special symbols are reserved words. "Real" cannot be a label in Burroughs ALGOL.

Spaces between characters play no part in ALGOL 60. The identifier, go to, is the same as goto and the special symbol go to is the same as goto. However, in some implementations of ALGOL such as Burroughs ALGOL for the B5500, spaces act as separators. In Burroughs ALGOL some is a single identifier while so me is a sequence of two identifiers; the first is "so" and the second is "me".

It is probably a good idea to avoid using spaces inside variables or constants. In this way our rules for constructing these components are in harmony with the other languages.

A2-3 Input-output statements

Now that you have become somewhat accustomed to the appearance of ALGOL characters and to their use in the elementary components of the language, like numerals, identifiers, and operators, you are ready to study the three important statement types, input, output and assignment.

You have already seen examples of input and output statements, namely:

```
read(A, B, C);
```

and

```
write(A, B, C, D);
```

The statements are simple in form. They always begin with the word

```
read
```

or

```
write
```

Following this we write, enclosed in parentheses, the list of variables whose values are either to be read or printed. If there is more than one, this list is separated by commas. We will not limit the number of list elements of an input or output list.

In this way we arrive at the general form of an input or output statement, i.e.,

```
read(input list);
```

or

```
write(output list);
```

For a read statement, the list elements are the variables whose values are to be assigned from the input data, while for a write statement the list elements are the variables whose currently assigned values are to be printed.

Strictly speaking, ALGOL 60 does not specify any particular form for the input or output statements or any particular standard name to suggest input or output. The forms we are describing here merely represent a typical ALGOL computer implementation. In some implementations, for instance, "print" is used in place of "write". While the names for the standard mathematical functions (Table A2-2) were chosen by those individuals who specified ALGOL 60, the identifiers like "read" and "write" are names of procedures chosen and defined by the persons who developed the particular implementation. These procedures are special programs which are made available automatically whenever

they are mentioned by name in an ALGOL program. Obviously, you should learn to use the procedure names and statement forms which are correct for the implementation you are using.

Executing a read statement

In this discussion we shall assume, as in the flow chart text, that input data originate on punched cards. The effect of executing a read statement is as follows:

1. First we assume a card is in position to be read by the computer's input device. If not, the execution of the program ceases immediately. In some, but not all, systems the computer might then print some message like

"YOU HAVE RUN OUT OF DATA"

or

"ALL INPUT DATA HAVE BEEN PROCESSED"

2. The contents of the card ready to be read is then transported to the computer memory, where it is examined one numeral at a time from left to right. A one-to-one match is then made between the numerals on the card and the variables of the input list with the result that each variable is assigned the matching value. What happens if the card fails to contain enough numerals to match all the variables of the input list or vice versa? This will depend on the nature of the read procedure, which is likely to vary with each ALGOL implementation. However, in most cases, another card will be read and numerals from this card matched with the as-yet-unmatched list elements. The process continues with as many cards being read as necessary until all the list elements have been assigned values. When this has occurred, we say that the list is exhausted. Execution of the read statement then terminates even if some numerals remain on the list data card that was being read.

At this point some examples can help us see how this matching process is carried out.

Example 1

Study Figure A2-6 where you see a read statement and a picture of a card that might possibly be read as a result of executing the given statement.

read (NUMBER, PAID, AMOUNT);

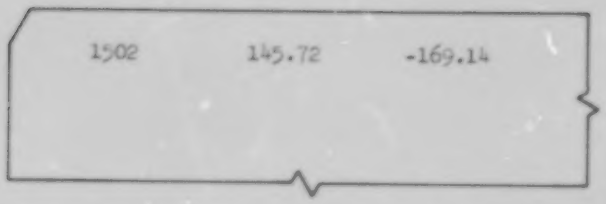
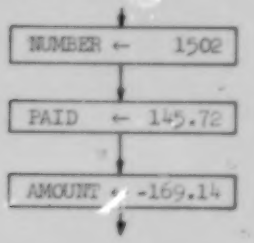


Figure A2-6. Picture of a data card and a read statement

The effect of executing this read statement is the same as if the following assignments had occurred successively.



Different read procedures may scan the contents of data cards in different ways. In the scheme suggested in Figure A2-6 the computer is expected to recognize the end of one numeral when it "sees" a space. Since there are many other schemes in use, you should learn the rules for punching data, often called formatting, which are used with your system. In our text we shall presume the system we are using is the very simple one we have just described.

Example 2

Study Figure A2-7 where you see a read statement along with three data cards which might possibly be read as a result of executing the given statement.

read (A, B, C, D, E, A, G, H, S);

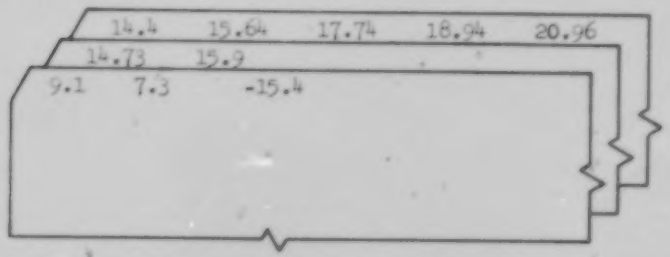
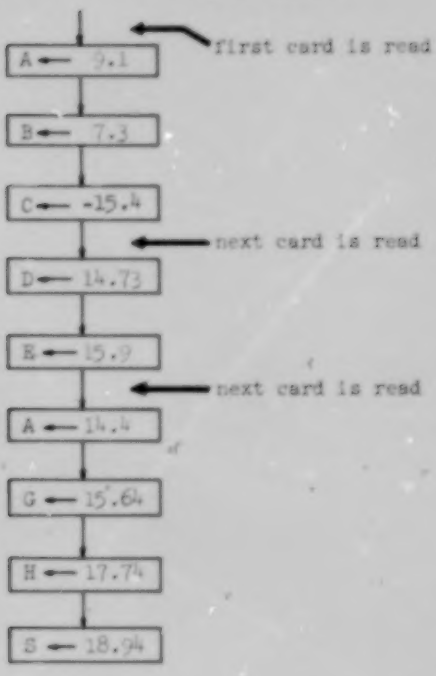


Figure A2-7. Picture of three data cards and a read statement

The effect of executing this read statement is the same as if the following nine assignments had been accomplished in order:



Notice that two assignments are made for A because it happens to appear twice, perhaps as a result of a key punch error. Following the principle of destructive read-in the current value for A at the end of the input step would be 14.4 and not 9.1 as perhaps desired.

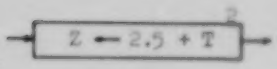
Notice also that input list is exhausted when 18.94 has been assigned to S. The last value on the last card, 20.96, is then ignored.

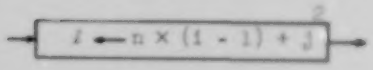
Exercises A2-3 Set A

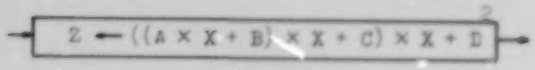
We imagine a class of very simple problems to be solved on the computer. Let the flow chart for each of these have a structure identical with the one in Figure A2-3. In the following exercises you are given box 2 in detail. Your job in each case is to decide what should be in box 1. Then

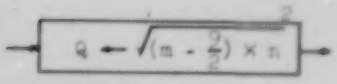
(a) write an appropriate read statement.

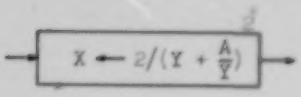
(b) Draw a picture of a typical card which could be read (in the system you will be using) as a result of executing the read statement which you have just written.

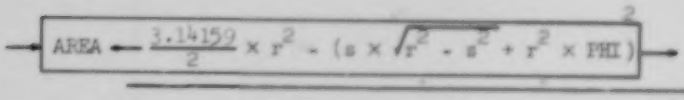
1. 

2. 

3. 

4. 

5. 

6. 

Executing a write statement

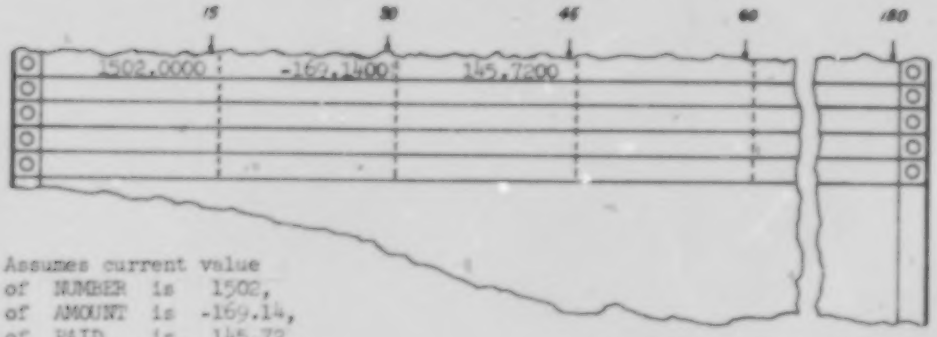
Writing is analagous to reading, but in a reversed sense. The value that is currently assigned to each element of the output list will appear typed or printed across the page by the output device. When each number is copied from its place in memory it is converted for printing from its internal representation to the desired external form.

You may have noticed that the write statement merely itemizes the variables whose values are to be written. It gives no clue as to how the numbers are to be displayed on printed paper. Write procedures in common use merely print the numerical values across the page, one for each item in the list--say up to m items per line. If there are more than m items in the list, additional lines will be printed, with up to m items on each succeeding line. We will normally assume m = 4 in our illustrations. The magic number m may vary depending on the width of the printer carriage that is used. You should consult your instructor for more details because some computer implementations have more elaborate write procedures than others, which would make it possible to obtain more elegant and more readable printed output. If this is the case with your computer, no doubt your instructor can furnish you with a booklet which adequately details the use of such fancy write procedures.

Example 1

Examine Figure A2-8 where you will see an example of a write statement and what it might accomplish when executed. Notice the uniform spacing of the numbers across the page, each number allotted the same width with the fractional part always given to four decimal places. How does the computer know to do this? The answer is simple. The write procedure arranges for this "formatted" appearance for us automatically. Write procedures vary from one implementation to another. Some, for instance, will print more decimal places, others fewer. These are details which are left for you to determine because they are related to the particular computer system you will be using.

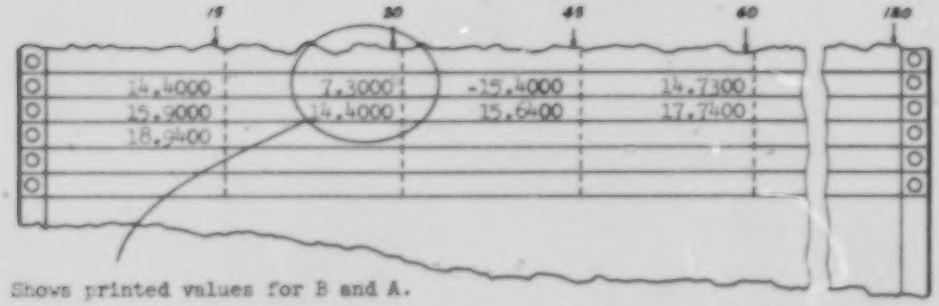
```
write (NUMBER, AMOUNT, PAID);
```



Assumes current value
of NUMBER is 1502,
of AMOUNT is -169.14,
of PAID is 145.72

Figure A2-8. Example of a write statement and a possible line of printed results caused by execution of the write statement

```
write (A, B, C, D, E, A, G, H, S);
```



Shows printed values for B and A.

Figure A2-9. When executed, the write statement shown here causes three lines to be printed. Since A appears twice in the output list, the first and sixth printed values are identical.

Example 2

Now suppose we wish to print, say nine values, as a result of executing a single write statement. With a write procedure controlling the printing of up to four items per line, we may expect output like that illustrated in Figure A2-9.

When the write statement is executed, values are copied out of memory one at a time from positions associated with A, B, C, etc. Each copied value is then converted for printing in the desired external form, as illustrated.

If a line printer device is used then, when four such numbers have been converted, a complete line is printed at once. However, if a typewriter device is used, each number is typed as soon as it has been converted to output form. This difference in behavior has no vital consequence as the net effect is the same. In any case, when four items have been dispensed with, the line is printed. Values are then copied from memory locations associated with four more list elements and printed. The last value is then copied from memory. At this point the procedure discovers that the list is exhausted, i.e., no more items remain to be copied. This discovery then signals the end of the process. Execution of the write statement is terminated and the computer is free to execute whatever statement happens to be next in the program.

Exercises A2-3 Set B

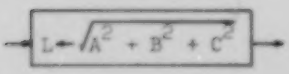
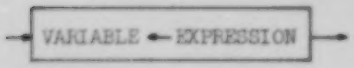
1 - 6. In these six exercises you will continue with the development you began in the preceding exercise set. For each your job now is to decide what should go inside box 3 of the flow chart. Then

(a) write a write statement which, if executed, would carry out the intent of box 3;

(b) for the data you used in the first set of exercises, compute the result which would be printed and show how it would appear on the printed page using the write procedure at your laboratory.

A2-4 Assignment statements

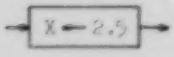
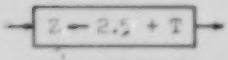
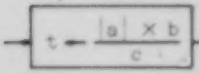
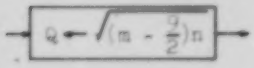
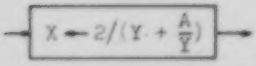
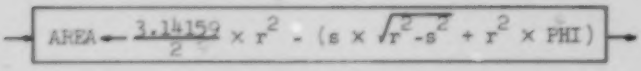
We shall first examine the parallel existing between our previously developed flow chart concepts of assignment steps and those of ALGOL statements.

	Flow Chart	ALGOL
specific example		L := sqrt(A↑2 + B↑2 + C↑2);
general form		variable := arithmetic expression

The ALGOL identifier is a character string built up of letters and digits (up to six in all, with the first character being a letter).

The ALGOL arithmetic expression corresponds to any meaningful computational rule which uniquely defines a single numerical value. There will be a few important restrictions to observe in writing such expressions correctly. Before considering these, let us see how several flow chart examples are rewritten in ALGOL.

Examples

	Flow Chart	ALGOL
1.		X := 2.5;
2.		Z := 2.5 + T;
3.		t := abs(a) × b/c;
4.		Q := sqrt((m - 9/2) × n);
5.		X := 2/(Y + A/Y);
6.		AREA := 3.14159/2 × r↑2 - (s × sqrt(r↑2 - s↑2) + r↑2 × PHI);

Rules for deciding the "type" of an evaluated expression

When we discussed and illustrated the step by step evaluation of arithmetic expressions in Section 2-4, we did not need to consider the questions of operand type. All our numbers were thought of as being available on a handy scratch pad. At each step in the evaluation if one operand of a binary pair happened to be an integer and another a number which had a fractional part, we simply took this in stride and added, subtracted, multiplied or divided as the operator symbol required. The computer evaluation of an ALGOL expression follows along similar lines, but what are these specifically?

To understand exactly what the computer does in each case, we need to realize that its action depends to some extent on which operator is involved. The operators fall into three classes and we will treat each separately.

class 1	+, -, ×
class 2	/
class 3	↑
class 4	↑

Class 1: +, -, ×

There are four possibilities because each operand may be either type real or type integer.

The rule is: The operation results in a number which is type integer if, and only if, both operands are integer. In all three other cases the result is of type real.

Let us see how we may project this concept in tabular fashion. Let the operation be depicted as

$$A \odot B$$

where A stands for the left-hand operand, $\odot$ is an operator symbol which in this case represents +, -, or ×, and B stands for the right-hand operand.

Table A2-4 shows at a glance the type of result obtained. In the table we have abbreviated real as R and integer as I.

Table A2-4

Types resulting from +, - or × operations

A \ B	R	I
R	R	R
I	R	I

Class 2: /

It is best to think of the division A/B as the multiplication of A by the reciprocal of B, or $A \times (B^{-1})$. When we do this, we see that the result of division is always of type real. Table A2-5 displays this case.

Table A2-5

Types resulting from / operation

A \ B	R	I
R	R	R
I	R	R

Class 3: ÷

The symbol ÷ is reserved for a special integer division, i.e., when both operands are integer, yielding a result of type integer, as shown in Table A2-6. The operation is not defined for any other type combination.

Table A2-6

Types resulting from ÷ operation

A \ B	R	I
R	-	-
I	-	I

Integer division is related to the greatest integer or "bracket" ([ ]) function, as can be seen from the following equivalences.

	ALGOL Expression	Computed Value	Mathematical Equivalent
1.	9 ÷ 10	0	[9/10]
2.	-10 ÷ (-10)	1	[10/10]
3.	11 ÷ 10	1	[11/10]
4.	10 ÷ 1	10	[10/1]
5.	-5 ÷ 10	0	-[5/10]
6.	-15 ÷ 10	?	-[15/10]
7.	10 ÷ (-1)	?	-[10/1]
8.	1 ÷ (-10)	?	-[1/10]

ALGOL's integer division yields a remainderless integer quotient with the same sign we would ordinarily obtain for the real quotient. For integers A and B it can be expressed mathematically as

$$A \div B = \text{TRUNK}(A/B) = \text{sign}(A/B) \times [|A/B|]$$

The right-hand side can be written in ALGOL as:

$$\text{sign}(A/B) \times \text{entier}(\text{abs}(A/B))$$

recalling that both the entier and sign function were defined in Table A2-2.

It is instructive to notice that for $A/B > 0$, $A \div B$ and $[A/B]$ are computationally the same but for $A/B < 0$ they are not.

Class 4: ↑

Finally, we have the operator ↑, denoting exponentiation such as A↑B. Many possibilities arise in this operation, depending on the sign of the A and the type and sign of B.

Exercises A2-4 Set A

1. Complete the table by filling in the type of the result, or "undefined".

Operands A	Operands B	Operation ⊙	type of result A ⊙ B
2	3	+	
6.3	.09	×	
100	.4	-	
6	4	/	
6	.4	+	

2. We wish to verify the mathematical formula for integer division in each of the eight ALGOL expressions in the preceding discussion. The first expression, 9 ÷ 10, is verified in the table shown below. Your job is to fill out the rest of the table for the remaining seven expressions, scanning each from left to right according to the step by step evaluation scheme developed in Section 2-5 of the main text.

Verifying that A ÷ B = sign(A/B) × entier(abs(A/B))

Case	A	B	① (A/B)	② sign(A/B)	④† abs(A/B)	⑤ entier(abs(A/B))	⑥ A ÷ B
1	9	10	0.9	+1	0.9	0	0
2							
3							
.							
.							
.							
8							

†What is step 3?

Exercises A2-4 Set B

Study the following cases in order to formulate rules to explain exponentiation. The rule has nine parts. Hint: The first 11 cases have to do with raising integers or real numbers to integer powers. Cases 12 - 22 deal with real powers only.

1.	$6 \uparrow 2$	means	6×6
2.	$-6 \uparrow 3$	"	$-6 \times 6 \times 6$
3.	$(-6) \uparrow 3$	"	$(-6) \times (-6) \times (-6)$
4.	$6 \uparrow n$	"	$6 \times 6 \times \ldots \times 6$ (n times)
5.	$2.5 \uparrow 3$	"	$2.5 \times 2.5 \times 2.5$
6.	$3 \uparrow 0$	"	1
7.	$4.25 \uparrow 0$	"	1
8.	$(-4.25) \uparrow 0$	"	1
9.	$3 \uparrow (-2)$	"	$1/(3 \times 3)$
10.	$3 \uparrow (-4)$	"	$1/(3 \times 3 \times 3 \times 3)$
11.	$2.5 \uparrow (-2)$	"	$1/(2.5 \times 2.5)$
12.	$5 \uparrow 2.0$	"	$\exp(2.0 \times \ln(5))$†
13.	$5.4 \uparrow 2.3$	"	$\exp(2.3 \times \ln(5.4))$
14.	$0 \uparrow 3.1$	"	0.0
15.	$0 \uparrow 0.0$	is undefined	
16.	$0 \uparrow (-514)$	" "	
17.	$(-5.2) \uparrow 3.2$	" "	
18.	$(-6) \uparrow 3.2$	" "	
19.	$(-6) \uparrow 0.0$	" "	
20.	$(-5.2) \uparrow 0.0$	" "	
21.	$(-16.1) \uparrow (-3.2)$	" "	
22.	$(-6) \uparrow (-3.2)$	" "	

†Recall Table A2-2. exp is the exponential function and ln is the natural logarithm function. The expression $\exp(2.0 \times \ln(5))$ means "e raised to a power which is 2 times the natural logarithm of 5." Natural logarithms, in case you haven't already seen them, use as their base the number e = 2.71828..., rather than the number 10. Using base 10, we mean the mathematical expression, $10^{2 \times \log_{10} 5}$.

Hint: let i be a number of type integer
let r be a number of type real
let a be a number of either type

Divide the rule into two parts. Let category 1, covering Cases 1 - 11 above, be represented by $a \uparrow i$. Now for $i > 0$, for $i = 0$, and for $i < 0$ try to specify the type of a^i. If you need to, break these cases further into subcases depending on the type of a. Next let category 2 cover Cases 12 - 22 above and be represented by $a \uparrow r$. Notice that here the relevant factor is the value of a. Is $a > 0$ or is $a = 0$ or is $a < 0$? Subdivide these cases if necessary until you can either specify that the result is undefined or you can determine the type.

Function references

We have already become acquainted with three of the basic components of arithmetic expressions, namely, constants, variables, and operators.

Two other important components are parentheses and function references.

In ALGOL we use parentheses in several different ways. In the expression

$$Z/(Y + A/Y)$$

the parentheses are used to form a subexpression that forces a desired ordering to the computation. In the expression

$$sqrt(a) \times b/c$$

parentheses are used in another way--to enclose the argument of a functio . A reference to a function value or a function reference, for short, consists of the name of the function followed by a pair of parentheses enclosing the argument. This convention in ALGOL is used for all functions.

Since some functions are given special marks or symbols, like $|\ |$, $\sqrt{\ }$, $\sqrt[3]{\ }$, which are not available in the ALGOL character set, it makes good sense to use the parentheses.

Here are some examples of typical function references:

```
sqrt(r↑2 + s↑2)

cos(PHI)

sin(6.4)

entier(3 + N)

abs(X)
```

They are easy to type from left to right as strings of characters. Remember the rule: Every function argument must be placed inside parentheses even if it is a single numeral or variable!

How are parentheses used in the expression

```
sqrt((m - 9/2) × n) ?
```

You can probably see that the outermost parentheses serve to enclose the argument of the sqrt function. This argument is the value which will be obtained upon evaluation of the expression

```
(m - 9/2) × n
```

in which the parentheses are employed for ordering computation.

As you can see, a function argument may itself be an ALGOL expression of decided complexity. In fact, in some of the expressions in the next exercise the argument of a function is expressed in terms of the value of another function. This is perfectly permissible in ALGOL. Another example of this is

```
exp(3 × ln(A))
    ~~~~~~~~~
argument of exp
```

Exercises A2-4 Set C

1. We would like to express $A^{3/2}$ in ALGOL, where $A \geq 0$. Keep in mind that $A^{3/2}$ is in this case the same as $\sqrt{A^3}$, or $(A^3)^{1/2}$. All of the following correctly express $A^{3/2}$ in ALGOL. Some are awkward. Some have superfluous operations. Comment on each and choose the one which appears to be the most efficient computationally.

   (a) abs(A↑1.5)
   (b) (A↑3)↑0.5
   (c) sqrt(A↑3)
   (d) abs(sqrt(A↑3))
   (e) (A↑1.5)
   (f) abs(A)↑1.5
   (g) sqrt(abs(A↑3))

2. If A can have negative values, which of the seven ALGOL expressions given in the preceding exercise correctly expresses $|A|^{3/2}$? If more than one, which is simpler computationally? Explain.

Unary minus

In Section 2-4 of the flow chart text you learned to distinguish between the unary and number-naming minus on the one hand, and the binary minus on the other hand. If a minus sign appears at the very beginning of an expression or immediately following a left parenthesis, it is either a unary minus or a number-naming minus. It cannot be a binary minus.

Here are some examples in ALGOL statements.

```
(1) Q := -5;
(2) Q := -(A + B);
(3) Q := -A↑(-C);
(4) Q := (-4)↑2;
(5) Q := -(4↑2);
(6) Q := -4↑(2);
(7) Q := -4↑2;
(8) Q := sin(-A + B × (-cos(C)));
```

Note that (4) assigns a value of +16 to Q. Remembering from Table 2-4 of the flow chart text that exponentiation takes precedence over unary minus, we see that (5), (6), and (7) each assign -16 to Q. Similarly, -A↑(-C) is the same as -(A↑(-C)). Two operators may not be written side-by-side. Thus, A × -B is invalid. We must write instead A × (-B) or perhaps -A × B. Similarly, A↑+ C or A↑- C are both invalid.

Exercise 2-4 Set D

Correct the following three invalid ALGOL statements. What problems arise in correcting the second and third statements?

(1) T := B × -A;

(2) F := C/-3 + 4;

(3) G := A + B × (C × -F/D);

A2-5 The order of computation in an ALGOL expression

We have not deferred this question because there is anything special to say about ALGOL that is different from what was already said in Chapter 2. On the contrary, the rules for the order of computation are precisely the same. Close the book and try to reconstruct them. Then compare with the following.

1. When parentheses are used to nest one subexpression inside another, evaluate the nested subexpressions from the innermost to the outermost.

2. Within any one subexpression evaluate in descending order of precedence:

Highest	function reference
↓	↑ (exponentiation)
	×, /, ÷
Lowest	+, -

3. In case of a tie in precedence level (within any subexpression) perform those operations in the tie from left to right.

A2-6 Meaning of assignment when the variable on the left is of different type from the expression on the right

Is it possible to convert a number from integer to real representation or vice versa? Notice that until now all the assignment statements we have illustrated were homogeneous in the sense that the variable on the left of the (:=) sign and the number evaluated from the expression on the right were both of the same type (real or integer). Two other obvious possibilities in ALGOL are both legal!

In short, we have four cases:

(a) real variable := real expression;
(b) integer variable := integer expression;
(c) real variable := integer expression;
(d) integer variable := real expression;

We have nothing further to say about cases (a) and (b). It is (c) and (d) we are interested in because such statements can be used to convert integers to reals and vice versa.

In case (c) the number assigned to the real variable simply has no fractional part.

Example

```
real T;
integer V, W;
V := 4;
W := 3 × 6;
T := V + W;
```

Observe that this sequence of ALGOL declarations and statements leads to a real value for T equal to 22.0. In other words, the integer sum of V + W results in the real value for T having a zero fractional part.

In case (d) the integer value assigned to the variable is rounded in the sense described in Section 2-5. In other words, when the expression is real and the variable is integer

variable := expression;

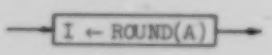
really means

variable := entier(expression + 0.5);

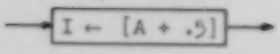
What this means is that if I is an integer variable and R is a real variable, the ALGOL statement

I := A;

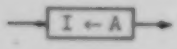
is equivalent to the flow chart assignment



or



Neither of these is identical to



Why? Because the flow chart variables I and A do not have specific digital representations associated with them. Hence, no rounding can be implied in simple flow chart assignment.

Example 1

```
real DIAM;
integer CIRCUM;
DIAM := 5.9;
CIRCUM := 3.14159 × DIAM;
```

This sequence of ALGOL statements leads to a value of 19 for CIRCUM, and not 18.535 which would be the actual circumference of a circle with a diameter of 5.9.

Example 2

```
integer OWE;
real BAL, WITHDL;
BAL := 50.97;
WITHDL := 92.49;
OWE := BAL - WITHDL;
```

Assuming BAL refers to bank balance, the overdraw is $41.52. The difference between BAL and WITHDL is -41.52. The rounded value assigned to OWE, i.e., entier(-41.52 + .5) = entier(-41.02) = -42.

Exercises A2-6

1. Which of the following statements are invalid ALGOL assignment statements? Explain.

   (a) T1 := T × VAR3/4.;

   (b) A := exp(A4•EXP↑3.0 + A3•EXP↑2.0);

   (c) Y := ln(sin + 4.0);

   (d) J := J + 1.0;

   (e) I := I/K + .4;

2. Assuming 1(b) above is valid, what changes would you propose in the interest of efficiency? If not, correct 1(b) and then improve it.

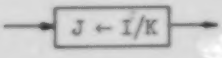
In each of the following, write a sequence of declarations and ALGOL assignment statements to accomplish the indicated task.

3. Assume the real variable V is assigned values which are always greater than or equal to zero. Assign to FPART, which is to be a type real variable, the fractional part of V.

4. Assign to INPT, an integer variable, the integral part of V. V is a real variable whose value is never negative.

5. Compare the following ALGOL statement with the accompanying flow chart box assuming I, J and K to be integers.

   J := I ÷ K;



   Are they the same? If not, draw another flow chart box to conform with the ALGOL statement.

6. Carry out the instructions of Problem 5 for the ALGOL statement

   J := I/K;

   and the same flow chart box.

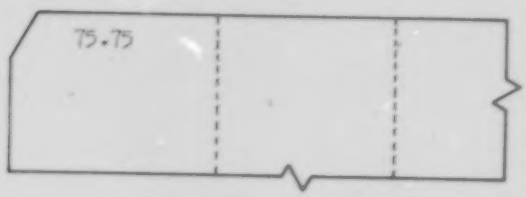
A2-7 Writing complete ALGOL programs

Remember the $20 bill problem? You are now able to write ALGOL statements to compute how many $20 bills are contained in (real) PRICE of your Jersey cow. Write your own version and then compare with that below.

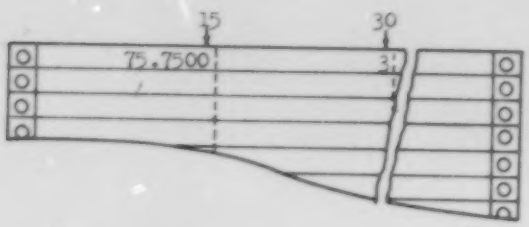
```
comment WHAT THE JERSEY COW WILL BRING;
integer NUM20 ;
real PRICE, RNUM20 ;
RNUM20 := PRICE/20.00;
NUM20 := entier(RNUM20);
```

A complete ALGOL program posesses a certain "structure". Before giving a formal description of this structure, we first form a complete program for the JERSEY COW problem as shown in Figure A2-10. We also put a label on the read statement and add a go to statement to make the program loop.

```
begin    comment WHAT THE JERSEY COW WILL BRING;
         real PRICE, RNUM20;
         integer NUM20;
 again:  read (PRICE);
         comment PRICE is given in dollars and cents;
         RNUM20 := PRICE/20.00;
         comment RNUM20 is the real number of twenty dollar bills;
         NUM20 := entier(RNUM20);
         comment NUM20 is the answer;
         write (PRICE, NUM20);
         go to again;
end
```



Picture of a data card



Picture of printed results

Figure A2-10. The Jersey Cow programmed in ALGOL

A complete ALGOL program must begin with a begin symbol and end with an end symbol. There are no exceptions. We might say that begin and end sandwich the declarations and statements. Notice that declarations precede the statements. We might think of the declarations as the "head" and the statements as the "body" of the program. A structure, then, begins to emerge as suggested in Figure A2-11.

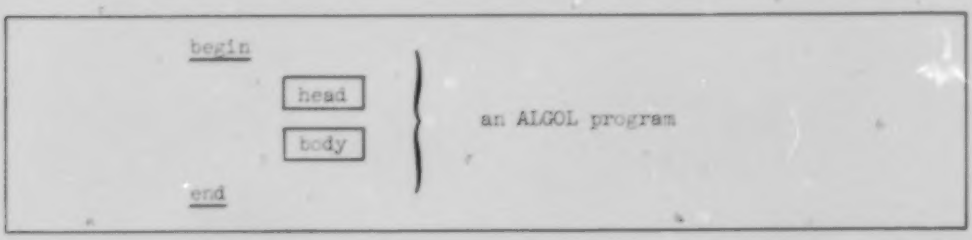


Figure A2-11. Structure of a complete ALGOL program

Now we introduce the concept of a "simple" and a "compound" statement.

The simple statement

We have seen in this chapter four examples of simple or basic statements. The basic forms were

```
variable := expression;
read (input list);
write (output list);
go to label;
```

Each of these is a request to a computer to carry out a specific set of actions and when completed, the next statement, usually in sequence, is executed. The exception is the go to statement.

The compound statement

It is perfectly proper to think of a sequence of simple statements as one statement since it describes a unit of action. In the flow-chart sense this group of individual actions might be thought of as a single box. We, therefore, employ the concept of a compound statement which consists of a sequence of statements preceded by "begin" and followed by "end". For our present purposes, the statements between begin and end will be considered to be simple statements. In Section A3-1 compound statements will be discussed further.

The general form of a compound statement can be thought of as

begin
S_1;
S_2;
.
.
.
S_n;
end

where S_1, S_2, etc., represent the individual statements.

The program

We create a program by packaging with the compound statement all the necessary declarations and comments that are necessary or appropriate. We then have as a general form for a program

begin
D_1;
D_2;
.
.
.
D_m;
S_1;
S_2;
.
.
.
S_n;
end

Here, the D's represent the one or more declarations which are now included to form the complete program. Comments may be inserted anywhere between the begin and end of any program.

In subsequent chapters we shall see many more examples of complete ALGOL programs. You will also be urged to write many of your own.

Exercises A2-7

1 - 6. In Section A2-3 you worked out the input and output details needed for six ALGOL programs each having the simple loop structure shown in Figure A2-3. You're to finish the job now by writing out on a coding sheet each of these six simple ALGOL programs.

7. What single assignment statement can replace the two that are used in the program given in Figure A2-10?

8. Recall the problem (in Section 2-5 of the main text) to simulate a carnival roulette wheel. It's presumed you have already drawn a flow chart for the situation specified. Now write a complete ALGOL program which is equivalent to your flow chart.

---

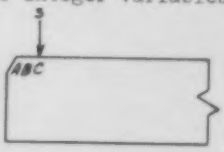
†A2-8 Alphanumeric data

An ALGOL program may deal with alphanumeric data, numeric data, or any mixture of these. When data is input the computer needs special instructions to convert alphanumeric data to the appropriate binary bit patterns for storing in memory. These patterns are of course different for storing alphanumeric characters than for numeric values.

Alphanumeric characters will be stored one character per word of memory. They may be stored in space that is earmarked for either integer or real variables. To avoid confusion, however, we shall always store such data in space (window boxes) for integer variables. A special input statement will be used when reading alphanumeric characters. Instead of the read statement we use a readsymbol statement†. The form is:

readsymbol(list of integer variables);

We illustrate in Figure A2-12.




```
integer X, Y, Z;
readsymbol(X,Y,Z);
```

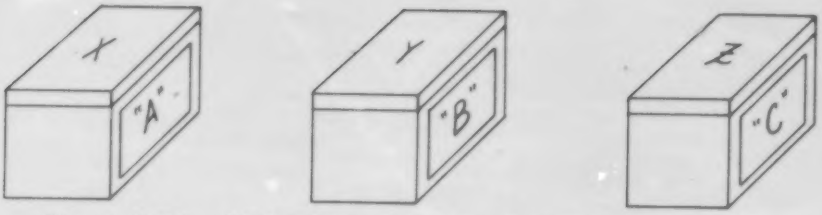


Figure A2-12. Window boxes after executing the readsymbol statement for the data card shown

Similarly, to output alphanumeric characters which may be currently assigned to certain variables, we use the printsymbol statement. For example,

printsymbol(Z, Y, X);

---

†Not all ALGOL implementations use exactly the same technique--you should consult your teacher or a reference manual to determine just how it is done on the computer system you are using. For instance, in Burroughs ALGOL alphanumeric values can be assigned to variables declared as type alpha. Three types are then possible: real, integer and alpha. The usual free field read or write procedures can be used with alpha variables. Such variables can have any set of six or fewer alphanumeric characters.

If this statement were executed following the input that was illustrated in Figure A2-12, the printed result would be Figure A2-13.

Figure A2-13. Printed result

Once an alphanumeric character has been assigned to a variable, it can be used on the right side of an assignment statement of the form

variable := variable;

Armed with this much information, the flow chart in Figure 2-28 can be written in ALGOL as shown here:

```
begin    comment utterly ridiculous process;
         integer A, B;
         readsymbol (A);
Box2:    readsymbol (B);
         printsymbol (A, B);
         A := B;
         go to Box2;
end
```

The only trouble with this program is that only the first character from each card is read into A or into B. After executing Box2 the first time using the data cards shown in Figure 2-28(c), A will have the value "M" and B the value "J". The rest of "MUTT" and "JEFF" is lost.

We can remedy this situation by a straightforward revision of the program, although by doing so we are forced to make the program quite a bit longer. To handle names of up to six characters we will use six variables in place of A alone, like A1, A2, A3, A4, A5, A6 and six more for B.

Here is the program:

```
begin    comment ridiculous process;
         integer A1,A2,A3,A4,A5,A6,
                 B1,B2,B3,B4,B5,B6;
         readsymbol(A1,A2,A3,A4,A5,A6);
BOX2:    readsymbol(B1,B2,B3,B4,B5,B6);
         printsymbol(A1,A2,A3,A4,A5,A6,B1,B2,B3,B4,B5,B6);
         A1 := B1;
         A2 := B2;
         A3 := B3;
         A4 := B4;
         A5 := B5;
         A6 := B6;
         go to BOX2;
end
```

A lot of work just to describe a ridiculous process--to be sure. Fortunately, techniques will be developed (Chapter 4) to make our ALGOL writing task far easier.

When a character, be it a digit or any other character, is enclosed in quote marks it will be understood to be an alphanumeric constant. Thus, the program

```
begin    integer A, B, C;
         A := "T";
         B := "-";
         C := "5";
         printsymbol (C, B, A);
end
```

will cause the output of the characters "5 - T".

Statements like

```
A := "T" + 2;
```

or

```
A := "2" + "2";
```

are meaningless and as far as we are concerned invalid. Statements like

```
A := "TF";
```

would also be considered improper because in our ALGOL system we are permitted to assign only one alphanumerical character to a variable. In other words

an alphanumeric constant can only be one character "long".

Example

Suppose the instructor who posed the original problem to compute

$$D = \sqrt{A^2 + B^2 + C^2}$$

now poses the problem this way:

> "Imagine that several sophomore geometry students have given you values of A, B and C, corresponding to the edges of a rectangular prism. You are to compute for each of them the distance D, which represents the length of a diagonal according to the formula suggested in Figure 2-1. Write a program which prints values of A, B, C, and the computed value for D, and then also prints for identification purposes the student's initials (three letters), his room number (a 3-digit integer) and seat location (a two-character code); like
>
> BJM
> 342
> C4

You will notice that all the identification can be thought of as alphanumeric including the room number--even though the instructor thought of it as an integer.

If identification of this sort were punched on a card, it might look like that shown in Figure A2-14.

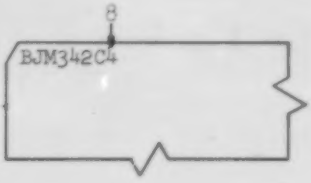


Figure A2-14. "ID" card

To input the data for this card we might use a readsymbol statement like

readsymbol(I1, I2, I3, R1, R2, R3, S1, S2);
(I1, I2, I3: initials; R1, R2, R3: room number; S1, S2: seat)

We will not need to change the structure of the algorithm in any significant way to achieve our new objectives--as you can see in Figure A2-15. We have merely added Boxes 0 and 4. The "ID" card is imagined always to precede the card bearing the student's values of A, B and C.

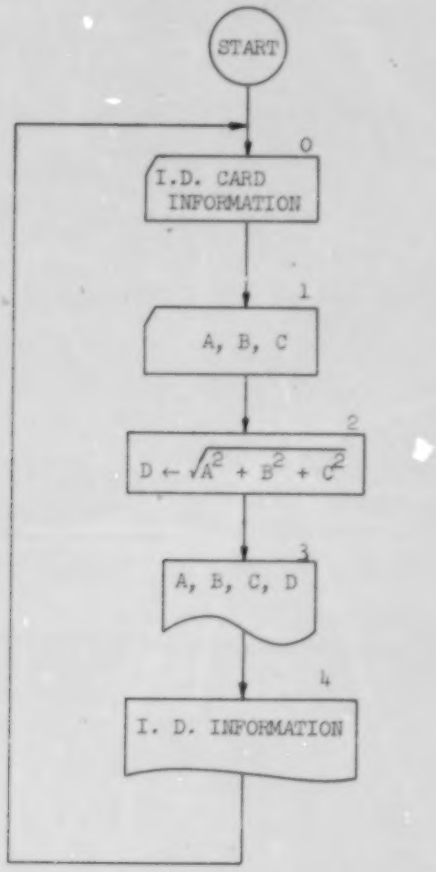


Figure A2-15. New flow chart

The corresponding ALGOL program--you could probably write it yourself--is shown in Figure A2-16.

```
begin    comment Evaluation of D;
         integer I1, I2, I3, R1, R2, R3, S1, S2;
         real A, B, C, D;
BOXZERO: readsymbol(I1, I2, I3, R1, R2, R3, S1, S2);
         read (A, B, C);
         D := sqrt(A↑2 + B↑2 + C↑2);
         write (A, B, C, D);
         printsymbol (I1, I2, I3);
         printsymbol (R1, R2, R3);
         printsymbol (S1, S2);
         go to BOXZERO;
end
```

Figure A2-16. The ALGOL program that goes with Figure A2-15

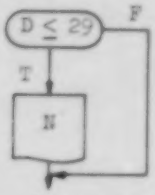
The three separate printsymbol statements are used in order to split up the "ID" information on to three successive printed lines.

Chapter A3

BRANCHING AND SUBSCRIPTED VARIABLES

A3-1 Conditional statements

In Section 3-1 of the flow chart text you studied techniques for branching by means of a condition box. In ALGOL you can write branching instructions by means of a conditional statement. Later in this chapter you will see that one ALGOL conditional statement can represent the equivalent of several condition boxes in the flow chart language. But first we will look at an ALGOL conditional statement that starts from one condition box--along with the two lines that emanate from it.



If D ≤ 29 then write (N);

Figure A3-1. Structure of one type of ALGOL conditional statement

The meaning of this conditional statement is clear: If D is less than or equal to 29, then write the value of N, otherwise, do not. The two flow chart branches then come together again.

In the flow chart language, the rules for what could be written inside the oval were not rigidly specified. As you might suspect, in ALGOL it is necessary to be rather specific about this matter, since a machine must translate your ALGOL program into executable code. An "if-clause" can be defined as a pair of arithmetic expressions separated by any one of the six relational symbols

$$< \quad > \quad \geq \quad \leq \quad = \quad \neq$$

and set off by if and then. The above simple form of conditional statement can therefore be written in the form

if-clause then	a read statement, a write statement, or an assignment statement

Recall the Muritanian Postal Regulations problem from Figure 3-1. This is one way to write the program in ALGOL:

```
begin    comment MURITANIAN POSTAL REGULATIONS;
         real A, B, C, D;
         integer N;
         BOX1: read(N,A,B,C);
               D := sqrt(A↑2 + B↑2 + C↑2);
               if D ≤ 29 then write (N);
               go to BOX1;
end
```

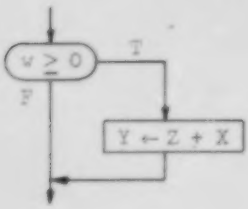
Exercises A3-1 Set A

Which of the following are correctly formed if-clauses? For those which are incorrect, cite the defects.

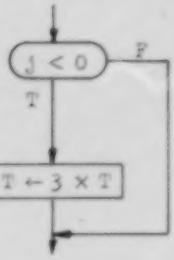
1. `if A < B < C then`
2. `if A + B < C then`
3. `if A + B > C + 1 then`
4. `if A + B > C; then`
5. `if C = B + 4 ≠ C then`

Write ALGOL statements for each of the following flow chart fragments.

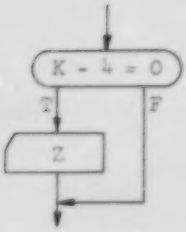
6.



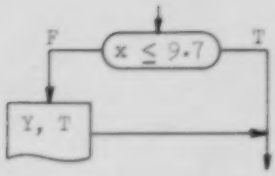
7.



8.



9.



Draw flow chart boxes for each of the following ALGOL statements. In each case rejoin the flow chart branches.

10. if Z < A then write (SS);

11. if sqrt(A↑2 + B↑2) = C then P:= P + 1;

12. if C + D ≠ T then read(A,B,N);

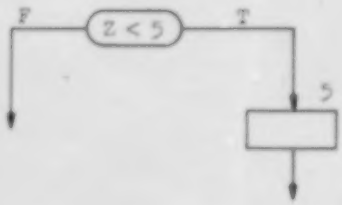
Actually, the statement which follows the "then" in the conditional statement need not be restricted to a single read, write or assignment statement. Any simple or compound statement can be used after then. For instance, the following ALGOL statements are proper:

```
if  x < 2  then
    begin y := z + x;
          T := 3 × T;
    end
```

In Section A2-7 we defined a compound statement as a sequence of statements; the sequence being preceded by "begin" and followed by "end". Now that we know what a conditional statement is, we can point out that the individual statements in the compound statement may be conditional or compound, as well as simple.

If the statement following then is a go to statement, then, of course, the two branches of the program do not rejoin immediately after the conditional. The following example is of this type:

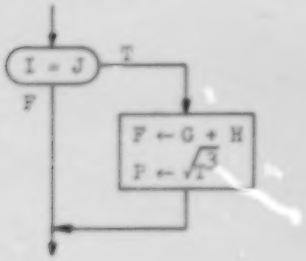
```
if  z < 5  then go to BOX5;
```



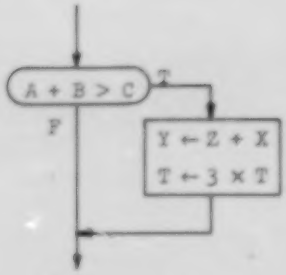
Exercises A3-1 Set B

Write ALGOL statements equivalent to each of the following flow chart fragments.

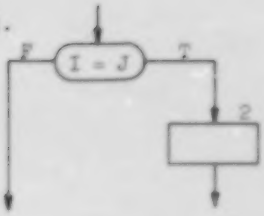
1.



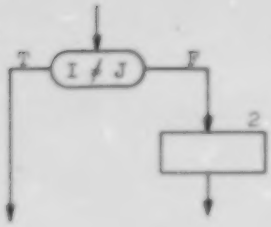
2.



3.



4.



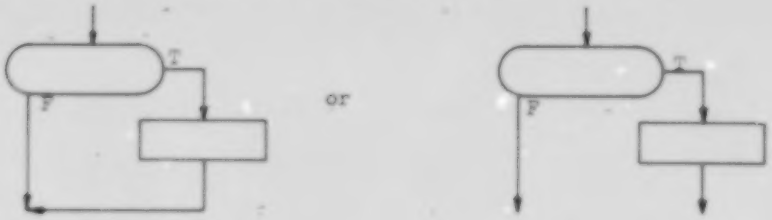
Draw flow chart boxes for the following ALGOL statements.

5. if Z ≥ A then go to BOX30;
   Z := A;

6. if P + 3 < S - K then go to BOX30;
   P := P × sqrt(P);
   S := S × sqrt(S);

7. if C + D ≠ T then
   begin F := 10 × T + 5;
   G := G - 5;
   end;

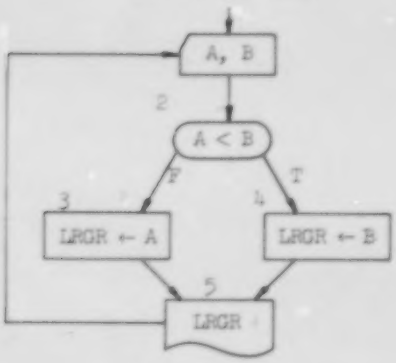
In the simple conditional statement the if-clause causes the program to branch and the part of the conditional following the word "then" affects only one of the branches. Whether the branches reunite immediately following the conditional depends on the topic of the particular conditional statement. The basic flow pattern for a simple conditional is then:



There is a second type of conditional statement which is equally useful and important. It makes use of an alternative when the if-clause condition is not fulfilled. The basic pattern for the "else" conditional is this:

Figure A3-2 repeats a familiar flow chart which lends itself to this second type of conditional statement. The ALGOL statements parallel the flow chart quite naturally.

Flow Chart Form




```
read(A,B);

if A < B then

LRGR := A else LRGR := B;

write(LRGR);
```

Figure A3-2. Conditional Statement, "Else"-type

Any statement type that could follow then in a conditional can legally follow else. That means any simple statement or compound statement can follow else.

You will recall that a compound statement is a sequence of ALGOL statements preceded by a begin and followed by an end. Since you can do practically anything with a compound statement, it is possible to insert the equivalent of a whole chunk of flow chart following the then and another whole chunk following the else.

Here are several examples of legal ALGOL conditionals:

```
if p < 0 then go to BOX1 else go to BOX5;
if k - r = 0 then X := Y - Z else
     begin y := z + x;
           T := 3 × T;
     end;
```

Notice that there is no semicolon before else. In fact, else must never be preceded by a semicolon!

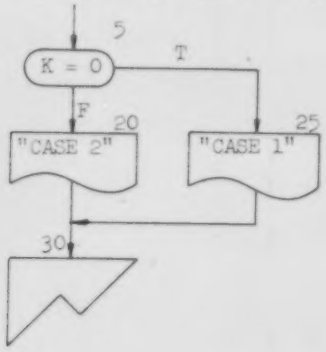
CAUTION: If you use a statement (whether compound or not) between then and else, you must observe the following restriction: The else may not be immediately preceded by a semicolon. Some other (non-blank) legal characters (e.g., end) must occur between them. For example, the following is strictly illegal:

```
if A > B then B := D; else B := D;
                     ↑
                get rid of it!
```

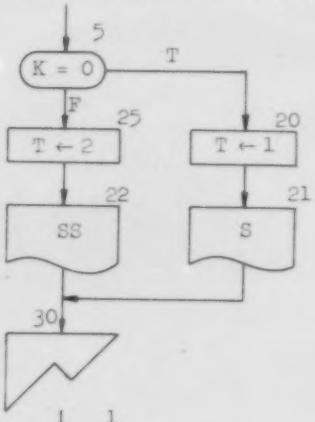
Exercises A3-1 Set C

For the following flow charts write ALGOL conditional statements. Use compound statements following then and/or else wherever feasible. Assume the next statement to be written corresponds with box 30, and is labeled BOX30.

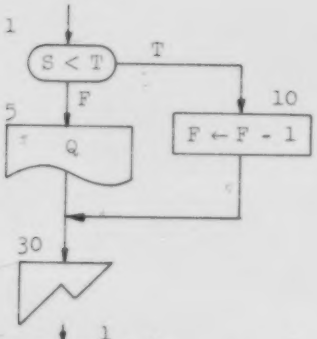
1.



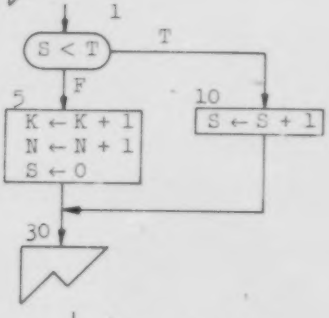
2.



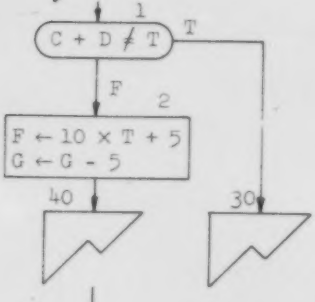
3.



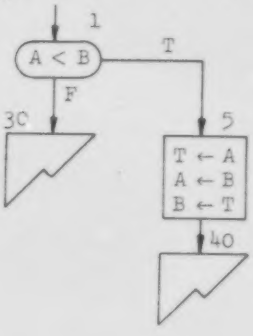
4.



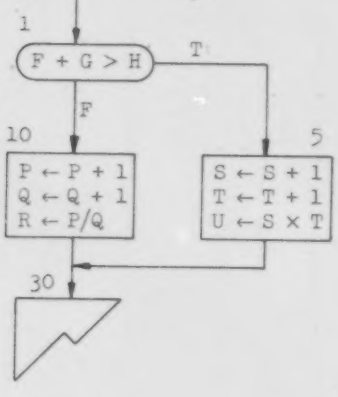
5.



6.



7.



A3

Tell which of the following are correct ALGOL conditional statements. For those which are not, tell what is wrong.

8. if A < B then T := T + 1; else T := T - 1;

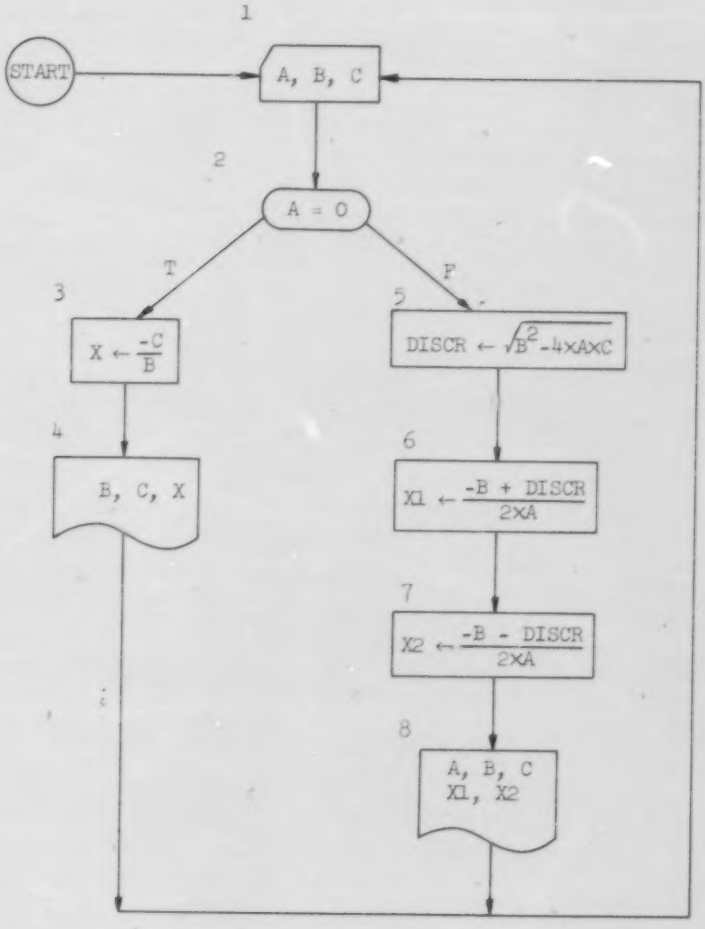
9. if COUNT < 100 else go to BOX40;

10. if COUNT + 1 then begin A := A × S; B := B × T; end;

11. if GLOOM < JOY then begin BOYS := BOYS + 1; GIRLS := GIRLS + 1;
else defeat := 1;

EXAMPLE: You are given that the equation $A \times x^2 + B \times x + C = 0$ has at least one real root, but you do not know that $A \neq 0$. Input A, B, and C. If $A = 0$, output B, C, and the root of the equation. If $A \neq 0$, output A, B, C, and the two roots (possibly equal) X1 and X2. Then come back to read A, B, and C for the next equation.

FLOW CHART SOLUTION:



ALGOL PROGRAM -- METHOD ONE:

```
begin     real A, B, C, X, X1, X2, DISCR;
          BOX1: read(A,B,C);
                if A = 0.0 then go to BOX3;
                DISCR := sqrt(B↑2 - 4.0 × A × C);
                X1 := (-B + DISCR)/(2.0 × A);
                X2 := (-B - DISCR)/(2.0 × A);
                write(A,B,C,X1,X2);
                go to BOX1;
          BOX3: X := -C/B;
                write(B,C,X);
                go to BOX1;
end
```

ALGOL PROGRAM -- METHOD TWO (using compound statements):

```
begin     real A, B, C, X, X1, X2, DISCR;
          BOX1: read(A,B,C);
                if A = 0 then begin X := -C/B;
                                    write(B,C,X);
                              end
                         else begin DISCR := sqrt(B↑2 - 4 × A × C);
                                    X1 := (-B + DISCR)/(2 × A);
                                    X2 := (-B - DISCR)/(2 × A);
                                    write(A, B, C, X1, X2);
                              end;
                go to BOX1;
end
```

Compare the methods used in the two ALGOL programs in the example. Which do you think is easier to read? To write? There is room for a difference of opinion here, as the choice between the two methods is largely a matter of taste. You should learn to read and write ALGOL in either style and use whichever seems more appropriate to a given problem.

In Section 3-1 of your flow chart text, you used output boxes involving not only variables but identifying remarks in quotes. In ALGOL we will handle identifying remarks in precisely the same fashion as you did in your flow chart work. We will merely enclose them in quotes and add them to the output list where appropriate. As with other output list items, a comma should separate the identifying remark from the following list item (if there is one) and from the preceding list item (if there is one). But this is nothing new, since you did it that way in flow chart output boxes.

Example 1

write("AFTER□", N, "□TRIES,□THE□ANSWER□IS□", X)

If the value of N were 3 and X were 4.9, then executing this write statement would result in printing the line

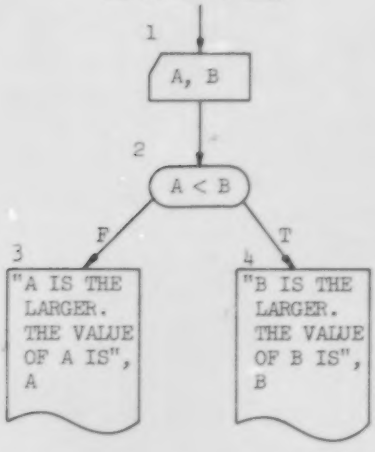
AFTER 3 TRIES, THE ANSWER IS 4.9
↑↑ ↑ ↑ ↑↑

one space above each arrow corresponding to each square (□) in the print statement. The square is simply our mark for a blank space; it isn't printed, and on a punched card it is represented by a column with no punches in it.

Example 2

Recall the flow chart from Figure 3-4. Figure A3-3 shows how this flow chart could be written in ALGOL.

Flow Chart Form



ALGOL Statements

```
read(A,B);
if A < B then write("B IS THE LARGER.  THE VALUE OF B IS", B)
         else write("A IS THE LARGER.  THE VALUE OF A IS", A);
```

Figure A3-3. Use of identifying remarks in ALGOL output

Strictly speaking, it isn't necessary that a write statement call for printing of any numerical values. One may wish to write only remarks or messages. For example:

```
write("A␣IS␣THE␣LARGER");
```

or

```
write("KILROY␣WAS␣HERE");
```

In Burroughs Algol any string of characters placed between quote marks may be used as a "field specification". The effect is to print the string without the quote marks in the line at this point. An example is:

```
write(< B IS THE LARGER" >);
```

The less-than symbol, "<", at the beginning signals the compiler that what follows is not a variable whose value is to be written, but some "format"

information. Format information includes literal strings as well as directions for how variables are to be printed. The free-field write procedure in Burroughs Algol does not allow the insertion of strings of identifying characters.

If A and B were integers of 5 digits or less in Figure A3-3, the Burroughs Algol for the statements would be

```
READ(A,B);
IF A < B THEN WRITE(< "B IS THE LARGER.  THE VALUE OF B IS", I5 >,B)
         ELSE WRITE(< "A IS THE LARGER.  THE VALUE OF A IS", I5 >,A);
```

I5 in the write statement above is a field specification which reserves 5 spaces at the place in the line where a variable is to be printed and causes the number to be printed as an integer. The greater-than symbol, ">", indicates the end of the format information. What follows is the variable to be printed.

The integer field specification has the form Iw where w is the field width.

Two other field specifications are fairly common. These are the F and E fields for printing real numbers. The F field specification is of the form F w.d where w denotes the field width and d denotes the number of decimal places to the right of the decimal point. The number will be rounded to d decimal places.

The third type has the form E w.d where again w is the field width and d the number of decimal places to the right of the decimal point. An example would be E20.10. The E-type is called floating-point type because the printed result includes a power of ten, adjusted to allow d places to the right of the decimal point.

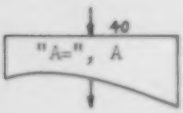
Formatting is a topic which has a great deal more detail than we need to include here. The Fortran language supplement to this course can be consulted for more information on format.

Exercises A3-1 Set D

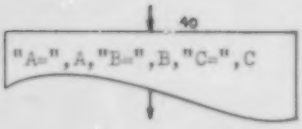
Write an ALGOL print statement for each of the following output boxes.

1.

2.



3.



"Nesting" one conditional statement within another

There is one more implication of the use of compound statements within a conditional statement that we should examine before going on. A compound statement may contain one or more conditional statements. Thus, a conditional statement may contain something that itself contains more conditional statements. The following ALGOL program for the flow chart in Figure 3-7 illustrates one way of using a conditional within a compound within a conditional:

```
begin     comment TALLYING;
          integer LOW, MID, HIGH, T, N, COUNT;
          read(N);
          COUNT := 0;
          LOW := 0;
          MID := 0;
          HIGH := 0;
BOX3:     read(T);
          if T ≤ 50 then LOW := LOW +1
                    else begin if T ≤ 80 then MID := MID +1
                                         else HIGH := HIGH +1;
                               end;
          COUNT := COUNT +1;
          if COUNT ≤ N then go to BOX3;
          write("VALUES OF COUNT, LOW, MID, AND HIGH ARE", COUNT,
                LOW, MID, HIGH);
end
```

Exercises A3-1 Set E

1 - 5. Refer to Exercises 9 - 13, Section 3-1, Set A in your flow chart text. For each of these five exercises you prepared a flow chart of a simple algorithm. Now, write an ALGOL program corresponding to each flow chart. Choose statement labels, in Exercises 12 and 13, to correspond with the box numbers used in the flow charts.

Exercises A3-1 Set F

1 - 6. Refer to Exercises 1 - 6, Section 3-1, Set B. For each of these six exercises you prepared a flow chart of a simple summing algorithm. Now, write an ALGOL program corresponding to each flow chart. Choose statement labels, where needed, to correspond with the box numbers used in the flow charts.

A3-2 Auxiliary variables

The use of auxiliary variables in ALGOL programs mirrors what you have already learned in the flow chart text.

Exercises A3-2 Set A

1 - 5. Refer to Problems 2 - 6 of Exercises 3-2, Set A in your flow chart text. Write ALGOL programs for each of the flow charts you have constructed for these problems.

You might like to see how the flow chart for the Euclidean Algorithm given in Figure 3-14 might work out in ALGOL:

```
begin comment EUCLIDEAN ALGORITHM for two non-negative integers;
      integer A, B, r;
      read (A,B);
      write("THE GCD OF", A, "AND", B, "IS");
      if A ≤ B
          then begin
BOX5:          if A = 0
                   then begin write(B);
                              go to HALT;
                        end
                   else begin r := B - entier(B/A) × A;
                              B := A;
                              A := r;
                              go to BOX5;
                        end;
               end
          else begin r := B;
                     B := A;
                     A := r;
                     go to BOX5;
               end;
HALT:
end
```

In this example you should note a different kind of basic statement: the dummy statement. It consists of no symbols at all and shows up only by means of its label, in this case "HALT". Nothing is to be done at this point. We merely want to mark the place at the end of the program so that we can use the statement

go to HALT;

Not all programs require this technique; but it is frequently useful when the box → (STOP) occurs in the flow chart and we want to express this in ALGOL.

Note that we must not forget to declare the type of any auxiliary variable that is used in the program. r is an auxiliary variable and appears in the integer declaration.

Exercises A3-2 Set B

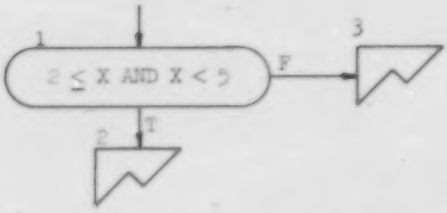
Write an ALGOL program corresponding to the flow chart you constructed in Problem 2 of Exercises 3-2, Set B of your flow chart text.

Exercises A3-2 Set C

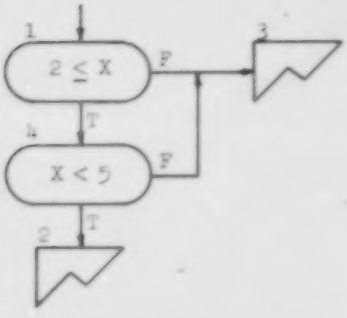
1 - 8. Refer to Problems 1 - 8, Exercises 3-2, Set C in your flow chart text. For each of these eight exercises you prepared a flow chart of a simple algorithm related to coordinates of points on a straight line. Now, write an ALGOL program corresponding to each of these flow charts. Choose statement labels, where needed, that are of the form BOX1, BOX2, etc., to correspond to the box numbering of the flow charts.

A3-3 Compound condition boxes and multiple branching

In Section 3-3 you encountered condition boxes involving more than one decision, e.g.,



and you saw that this single decision box was equivalent to a pair of boxes



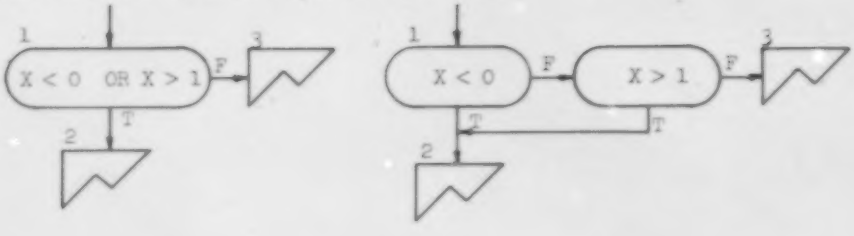
From your study of Section A3-1 you know that boxes 1 and 4 in the latter flow chart can be expressed in ALGOL as

```
BOX1:  if 2 ≤ X then begin if X < 5 then go to BOX2;
                     end
       else go to BOX3;
```

or, if the statement BOX3 immediately follows the statement BOX1

```
BOX1:  if 2 ≤ X then begin if X < 5 then go to BOX2;
                     end;
```

Similarly, the following pair of flow charts



may be written in ALGOL as

```
BOX1:  if X < 0  then go to BOX2
                 else begin if X > 1 then go to BOX2;
                      end;
```

provided, of course, that the statement BOX3 immediately follows the statement BOX1.

In Section A3-1 you learned that any basic or compound statement was legal following then and also legal following else. In addition to basic and compound statements, conditional statements are also legal following else! Thus, we can abbreviate the last conditional statement BOX1 displayed above by

```
BOX1:  if  X < 0 then go to BOX2
                 else if X > 1 then go to BOX2;
```

> CAUTION: Conditional statements are not legal following then unless they are buried within a compound statement enclosed by begin and end. Thus, we can legally write
>
> ```
> A:  if X < 0 then go to BOX2
>              else if X > 1 then go to BOX2;
> ```
>
> but we cannot legally write
>
> ```
> B:  if 2 ≤ X then if X < 5 then go to BOX2;
> ```
>
> In place of statement B we must write
>
> ```
> C:  if 2 ≤ X then begin if X < 5 then go to BOX2;
>                  end;
> ```
>
> In other words, then if is forbidden but else if is alright.

Exercises A3-3 Set A

Which of the following are legal in ALGOL? For those which are legal, draw a flow chart. For those which are not legal, tell why they are not legal. Assume that a statement labeled BOX4 immediately follows the ALGOL code given in the exercise.

A3

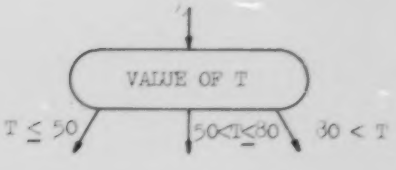
1. if A < B then go to BOX5 else go to BOX7;
2. if A < B then go to BOX5;
3. if A < B then if C < D then go to BOX5;
4. if A < B then go to BOX5 else if C < D then go to BOX7;
5. if A < B then go to BOX 5 else begin if C < 0 then go to BOX7; end;
6. if A < B then begin if C < D then go to BOX5; end;
7. if A < B then go to BOX5 else A := B;
8. if A < B then A := B;
9. if A < B then begin C := A; A := B; B := C; end;
10. if A < B then C := A; A := B; B := C; else go to BOX5;
11. if A < B then C := A; A := B; B := C; go to BOX5;

Exercises A3-1 Set B

1 - 7. Refer to Exercises 1 - 7, Section 3-3 in your flow chart text. For each of these exercises write ALGOL statements equivalent to the flow charts you prepared. For example, the flow chart example might be coded in ALGOL as:

```
        if X1 < X2 then go to BOX2 else go to BOX30;
BOX2:   if P > G then go to BOX20
        else if T = S then go to BOX20
        else go to BOX30;
```

Another topic you studied in Section 3-3 was multiple branching, for example



The ALGOL program we wrote earlier in this chapter corresponding to the tallying flow chart of Figure 3-7 used a succession of two two-way branches to achieve the effect of a three-way branch. The ALGOL statement was this:

```
if T ≤ 50 then LOW := LOW + 1
          else begin if T ≤ 80 then MID := MID + 1
                               else HIGH := HIGH + 1;
               end;
```

Compare this with the ALGOL statement for the single three-way branch of the flow chart in Figure 3-20.

```
if T ≤ 50 then LOW := LOW + 1
          else if T ≤ 80 then MID := MID + 1
                         else HIGH := HIGH + 1;
```

The only difference is the omission of "begin" and "end" around the second conditional statement.

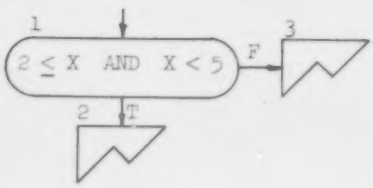
The warnings in Section 3-3 concerning the non-overlapping and exhaustive nature of the exit conditions from a condition box are not really necessary in ALGOL, since in an if ~~~ then ~~~ else if ~~~ then else if ~~~ then . . . statement

(1) the first if-clause that comes out true satisfies the condition of the entire if, so the remaining if-clauses are ignored.

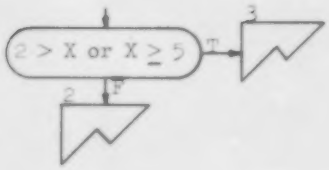
(2) If none of the if-clauses come out true and there is no else after the last then, we merely go on to the next piece of code after the conditional statement.

You may have noticed that the discrimination between what can follow then and what can follow else seems at first glance to be a bit "unfair" to compound condition boxes involving "and". If you didn't notice this, look at the first two examples in this section--the ones involving "and" and "or" and see which is easier to code in ALGOL.

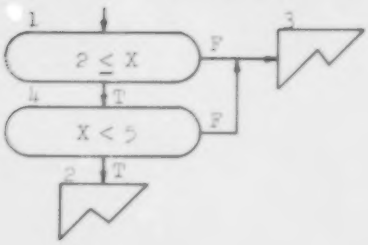
For relatively uncomplicated compound condition boxes such as



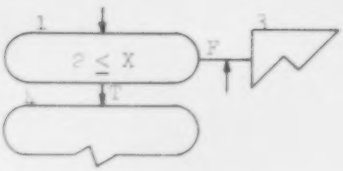
this "discrimination" actually poses no difficulty. If you know a little about the logic of the words "and" and "or" you may see without difficulty that we can replace the "and" box above by the following "or" box:

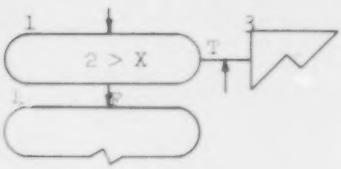


This may be easier to see by looking at the pair of condition boxes

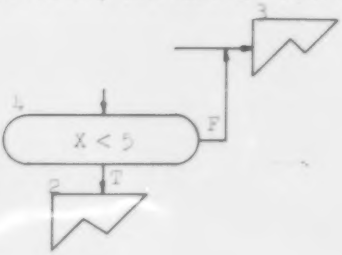


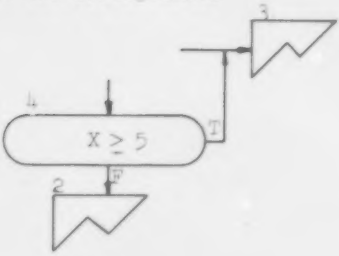
You will remember from Section 3-1 that the two arrangements



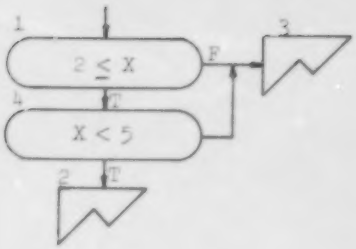


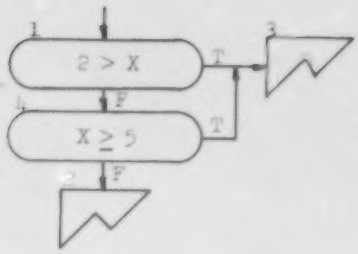
are equivalent, and that the same applies to the two arrangements



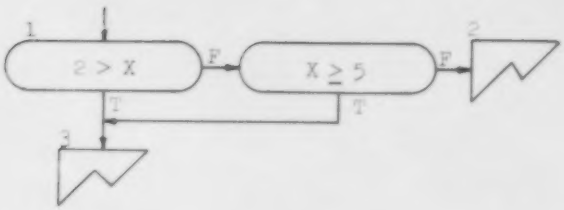


If we put these fragments together we find that the arrangements

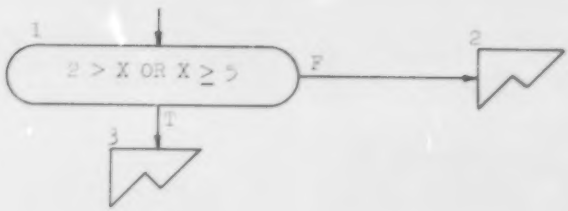




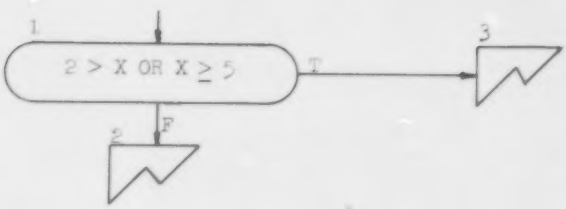
are equivalent. But now take a fresh look at the right-hand flow chart. By merely rearranging the drawing on the page we get



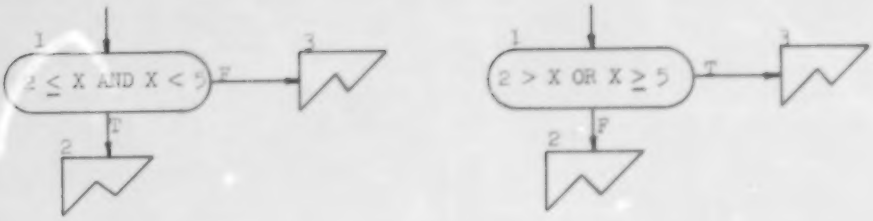
which, according to your study in Section 3-3, is the same as



But this is just another way of arranging the drawing for



Thus, you can see that the following two are equivalent condition boxes:

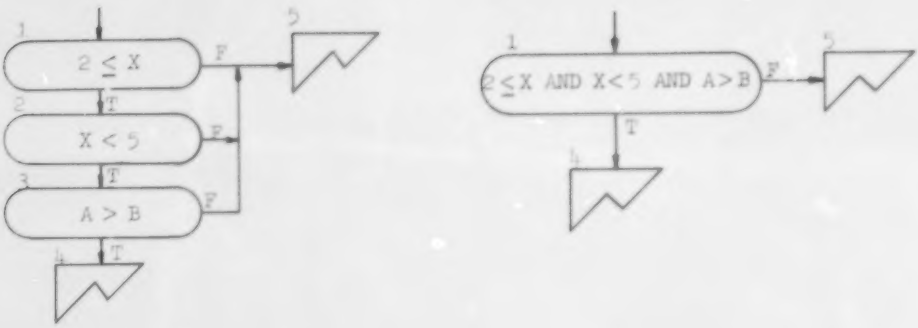


For the purpose of your work with ALGOL, the important lesson for you to learn from the foregoing discussion is that it is possible to transform a sequence of condition boxes connected by "T" arrows into a sequence of condition boxes connected by "F" arrows, and vice versa. This is important because of the correspondence between the ALGOL construction

if ~~~ then ~~~ else if ~~~ then ~~~ else if ...

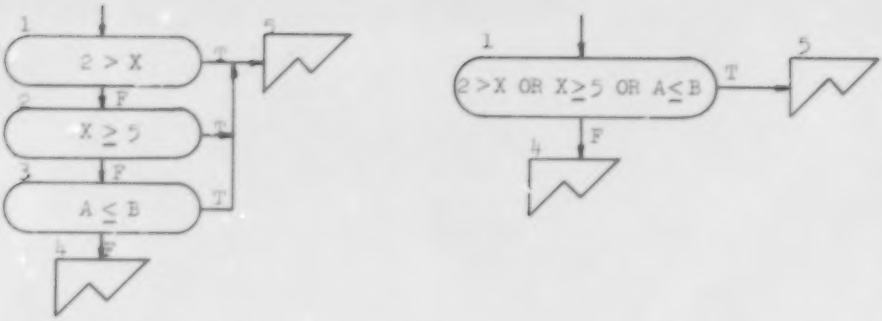
and a sequence of condition boxes connected in "series" by "F" arrows.

The algorithms illustrated below in Figures A3-4 and A3-5 are equivalent.




```
if 2 ≤ X then begin if X < 5 then begin if A > B then go to BOX4
                else go to BOX5; end; end;
```

Figure A3-4. T-series method of coding 2 ≤ X and X < 5 and A > B



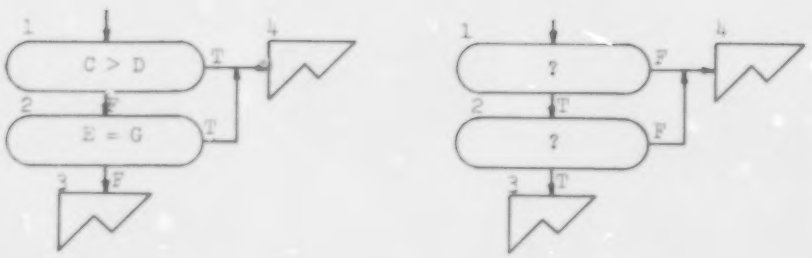

```
if 2 > X then go to BOX5
        else if X ≥ 5 then go to BOX5
                else if A ≤ B then go to BOX5
                        else go to BOX4;
```

Figure A3-5. Easier F-series way of coding the problem in Figure A3-4

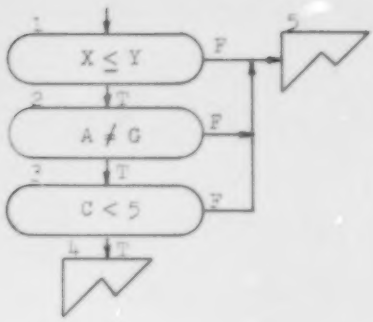
Exercises A3-3 Set C

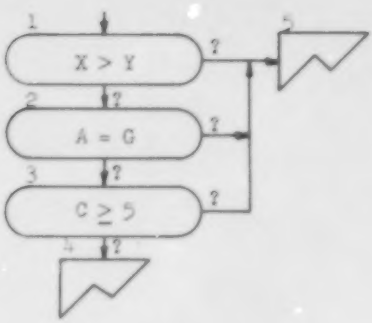
In Exercises 1 - 3, complete the flow chart on the right making it equivalent to the one on the left. Then write a single ALGOL conditional statement equivalent to either flow chart form. Is it easier to write the ALGOL from the first form or the second?

1.

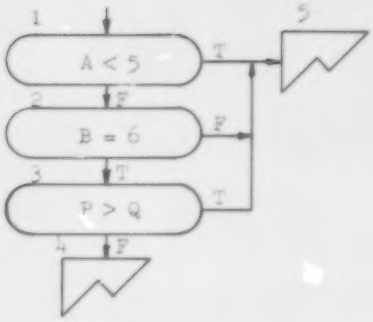


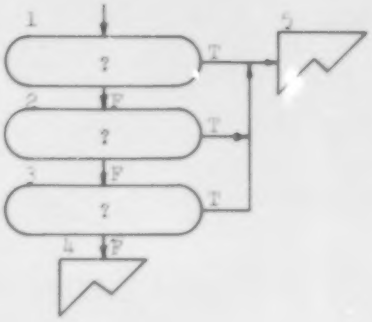
2.





3.





In Exercises 4 - 5 write an equivalent ALGOL statement without using begin or end. Hint: Use if ~~ then ~~ else if ~~.

4. if P < Q then begin if Q > R then go to BOX7 else go to BOX8; end;

5. if P = Q then begin if Q = R then begin if R ≠ S then go to BOX7 else go to BOX8; end; end;

In Exercises 6 - 7 write an equivalent ALGOL statement without using the else if construction.

6. if A ≠ B then go to BOX6 else if C = D then go to BOX6 else go to BOX7;

7. if A ≥ B then go to BOX6 else if B ≥ C then go to BOX6 else go to BOX7;

Draw a multiple branch condition box for each of the following ALGOL statements. Observe the cautions concerning non-overlapping and exhaustive exits as set down in Section 3-3.

8. if X ≤ B then go to BOX6 else if X ≤ B + 5 then go to BOX7 else go to BOX8;

9. if A = B then go to BOX6 else if A = C then go to BOX7 else go to BOX8;

10. Write an ALGOL statement for the following multiple branching condition box:

Exercises A1-3 Set D

1. Write an ALGOL program corresponding to the flow chart you prepared for Problem 10, Exercises 3-3.

2. Write an ALGOL program corresponding to the new flow chart which you prepared for the carnival wheel problem as the answer to Problem 11, Exercises 3-3.

A3.4 Precedence levels for relations

In ALGOL the precedence level for relation symbols mirrors that in your flow chart language and should require no special study.

A3-5 Subscripted variables

Representation of subscripted variables in ALGOL

Figure A3-6 shows how subscripted variables are represented in ALGOL.

Flow Chart Form	ALGOL Form
X_1	X[1]
X_N	X[N]
B_{I+1}	B[I + 1]
$Z_{5\times I+J+2\times K}$	Z[5 × I + J + 2 × K]

Figure A3-6. Representation of subscripted variables in ALGOL

As you can see from the figure, subscripted variables are represented in ALGOL by enclosing the subscript in square brackets and writing it following the variable to which the subscript is affixed. This is another example of a notation that enables ALGOL code to be written "on the line". Other examples you have seen include "A↑2" for "A^2" and "sqrt(X)" for "$\sqrt{X}$". Since the "greatest integer function" is represented in ALGOL by entier(X), there need be no confusion between use of square brackets for this function in flow chart language and its use for subscripts in ALGOL. In ALGOL square brackets always indicate subscripts. They are not available as a special type of parenthesis as in ordinary algebra. For parentheses you must always use "(" and ")".

Exercises A3-5 Set A

Write each of the following subscripted variables in ALGOL form:

1. X_5 2. Z_N 3. $CHAR_I$ 4. B_{I+2}

Allocation of memory storage for arrays

As you know, the computer must have a storage location available corresponding to each variable in a given ALGOL program. If all variables were of the form

A, X, CHAR, X_4, etc.

this would be a simple problem indeed. The processor could merely assign a storage location to each variable occurring in your program.

But what about X_I? How can the processor know for what values of I to assign storage locations? It cannot tell merely by looking at the occurrences of X_I in your program. But if it were to wait until the program was being executed, it might find that it needed locations for X_1, X_2, X_3, ..., X_{25} and had assigned locations only for X_1, X_2, ..., X_{10}. Since storage of arrays such as X_1, X_2, ..., X_{25} in consecutive locations in storage is of importance in efficiency of program execution in most computers, we would like to have advance knowledge of the range of values possible for a subscript before we start executing the portion of the program in which that subscript occurs. More precisely, we require at least knowledge of the maximum range so that we will allow enough locations.

In ALGOL this problem is solved by means of a declaration. You have already learned about the declarations real and integer. For arrays (subscripted variables) we have the declarations real array and integer array. Figure A3-7 shows the form of the array declarations and compares them to the declarations real and integer.

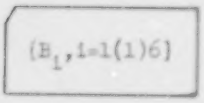
Declaration	Resulting Storage Allocation
real X;	one location for X
integer I, J;	one location for I and one for J
real array X[1:5];	five locations: one each for X_1, X_2, X_3, X_4, and X_5
integer array I[5:60];	56 locations: one each for I_5, I_6, ..., through I_{60}
real array X[-2:3];	six locations: one each for X_{-2}, X_{-1}, X_0, X_1, X_2, and X_3
real array X[1:3], Y[2:5];	one location each for X_1, X_2, X_3, Y_2, Y_3, Y_4, and Y_5

Figure A3-7. Storage allocations for ALGOL declarations.

Input and output of arrays

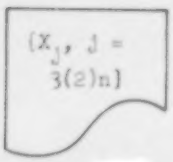
You will study more convenient methods for input and output of arrays in Chapter 4. For the present, we shall use a loop to input an array. Figure A3-8 illustrates one possible method.

Flow Chart Box



Crude ALGOL Program

```
        I := 1;
AGAIN:  read(B[I]);
        I := I + 1;
        if I ≤ 6 then go to AGAIN;
```




```
        J := 3;
REPET:  write(X[J]);
        J := J + 2;
        if J ≤ n then go to REPET;
```

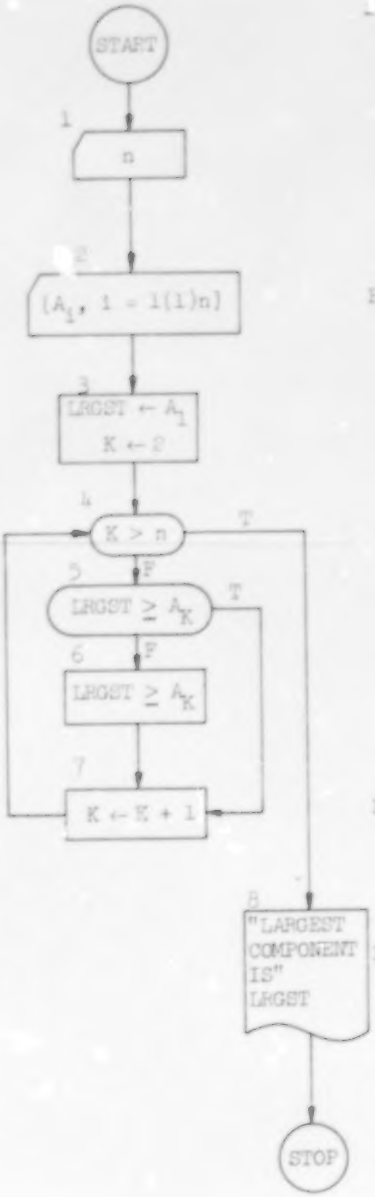
Note that n will already have been assigned a value before the array is written out in the second example.

Figure A3-8. Crude method for input and output of arrays in ALGOL

Example

Draw a flow chart and write an ALGOL program to find largest component of an n-component vector ($n \leq 100$):

Flow Chart



ALGOL

```
begin  comment LARGEST COMPONENT;
       real array A[1:100];
       real LRGST;
       integer N, K, I;
       read(N);
       I := 1;

BOX2:  read(A[I]);
       I := I + 1;
       if I < N then go to BOX2;
       LRGST := A[1];
       K := 2;

BOX4:  if K > N then go to BOX8;
       if LRGST ≥ A[K] then go to BOX7;
       LRGST := A[K];

BOX7:  K := K + 1;
       go to BOX4;

BOX8:  write("LARGEST COMPONENT IS",
              LRGST);

       end
```

Exercises A3-2 Set B

In Exercises 1 - 3 write the necessary ALGOL declarations and ALGOL statements to input the real arrays indicated. In each case assume that before we start, n or k has been assigned a specific value. Use the inequality on the right only for the purpose of reserving space, i.e., in the array declaration.

1. $(A_I,\ I = 1(1)k)$ Assume $k \leq 50$.

2. $(B_J,\ J = 5(2)n)$ Assume $n \leq 125$.

3. $(A_I,\ I = 10(1)n),\ (B_I,\ I = 10(2)n)$ Assume $n \leq 50$.

Exercises A3-3 Set C

1. Write an ALGOL program corresponding to the flow chart in Figure 3-24(b) (the carnival wheel problem using subscripts).

2. Write an ALGOL program that corresponds to Figure 3-25.

3. Write an ALGOL program corresponding to the flow chart you drew for Problem 8 of Section 3-9, Set A. Assume the value of n will never be greater than 50. Assume polynomial coefficients are punched in order on successive data cards.

Exercises A3-3 Set D

Write an ALGOL program for the flow chart you drew for part c of Exercise 3-9, Set C in the main text. Assume by "any size orchestra" we mean one that has no more than 125 players. Assume the ages of the players are integers punched on successive cards.

A3-6 Double subscripts

Representation of double subscripts in ALGOL

In Section A3-5 you learned that a subscripted variable like

$$x_i$$

could be written in ALGOL as

```
X[I]
```

Now, in Section 3-6, you have been introduced in your flow chart language to doubly subscripted variables such as

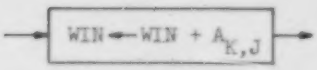
$$A_{I,J}$$

The ALGOL representation for such a doubly subscripted variable is just about what you would expect it to be:

```
X[I,J]
```

The two subscripts are separated by a comma.

Thus, the assignment box



could be written in ALGOL as

```
WIN := WIN + A[K,J];
```

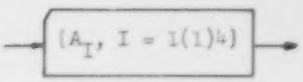
Allocation of storage for doubly subscripted arrays

An array declaration is required to allocate space for doubly subscripted variables. You can no doubt guess how this is done. For example,

```
real array A[1:3, 1:4];
```

declares that acceptable values of the first subscript on A lie between 1 and 3, inclusive, and of the second subscript, between 1 and 4, inclusive.

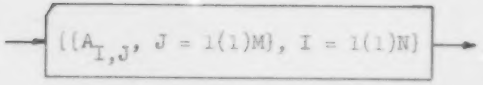
In Section A3-5 you learned to write the ALGOL program for an input box such as



as

```
        I := 1;
THERE:  read(A[I]);
        I := I + 1;
        if I ≤ 4 then go to THERE;
```

It is possible to generalize this technique to take care of input boxes such as



but to do this in Chapter A3 would cause you busy work which can be avoided when we have more powerful tools for looping that you will learn in Chapter A4. For now, merely place a comment in your ALGOL program showing where the input (or output) of a doubly subscripted array is to occur. Then, in Chapter 4, you will learn how to complete the program to run on a computer.

With this convention, we are now prepared to look at an ALGOL program for the problem in Figure 3-34.

```
begin   comment GAME;
        real array A[1:6, 1:6];
        real WIN, LOSE, NET;
        integer I, J, K, L;
          comment INPUT OF ARRAY A[I,J] GOES HERE;
        read(K,L);
        WIN := 0;
        LOSE := 0;
        I := 1;
        J := 1;
BOX4:   if J ≤ 6 then go to BOX5 else go to BOX6;
BOX5:   WIN := WIN + A[K,J];
        J := J + 1;
        go to BOX4;
BOX6:   if I ≤ 6 then go to BOX7 else go to BOX8;
BOX7:   LOSE := LOSE + A[I,L];
        I := I + 1;
        go to BOX6;
BOX8:   NET := WIN - LOSE;
        write(NET);
end
```

Exercises A3-6

1 - 5. Refer to Exercises 1 - 5, Section 3-6. For each of these exercises you have already drawn flow charts describing certain "row-" or "column-operations" on a matrix. Now your job is to write equivalent ALGOL statements preceded by all necessary declarations for each of these partial flow charts.

Chapter A4

LOOPING

A4-1 The "for clause" and the "for statement"

It should come as no surprise that the remarkable box we called the "iteration box" has a perfect parallel in ALGOL. This shorthand is called the "for clause." An example is shown in Figure A4-1.

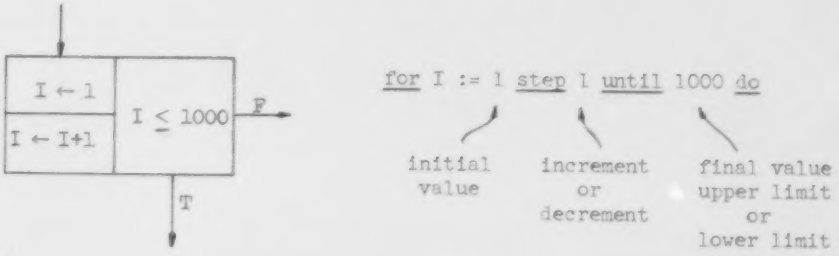


Figure A4-1. The iteration box and an equivalent for clause

An entire flow chart loop consisting of the iteration box and the computation portion "hung" on it can be expressed in ALGOL with the for statement. We present this parallel in Figure A4-2. The algorithm displayed, you will recognize, is the Fibonacci Sequence generator. (Figure 4-6)

The "for statement" consists of a for clause followed by a single statement, either simple or compound, that is to be repeated.

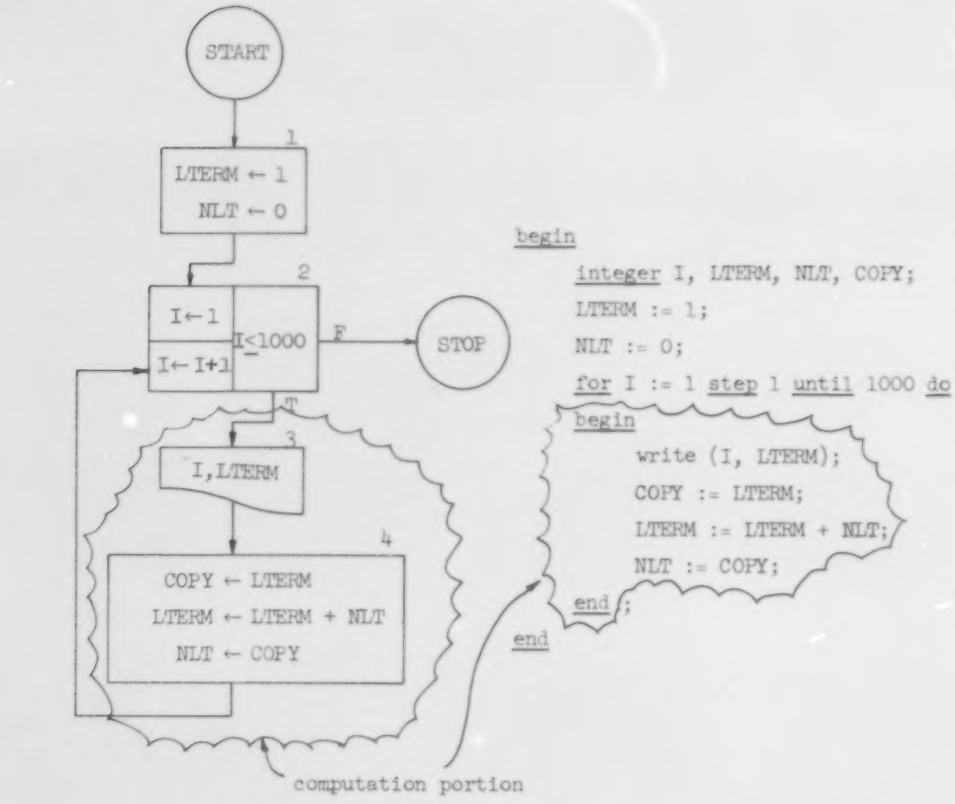
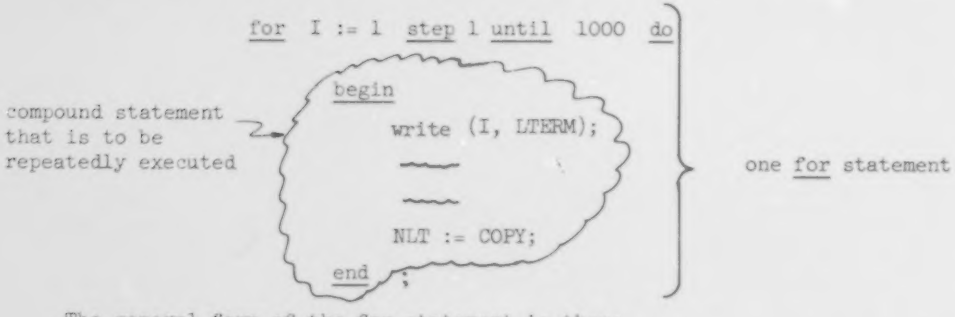
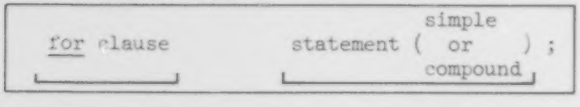


Figure A4-2. The flow chart loop and the equivalent ALGOL for statement

In the example shown the whole for statement is:


for I := 1 step 1 until 1000 do
begin
write (I, LTERM);
NLT := COPY;
end;
compound statement that is to be repeatedly executed
one for statement

The general form of the for statement is then:



The form of the for clause is seen to be:

for LOOP COUNTER := INITIAL VALUE step INCREMENT
until FINAL VALUE do

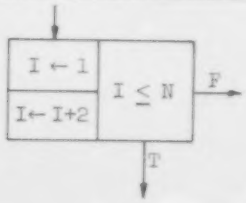
We mean to imply here that if the INCREMENT is a positive quantity, the FINAL VALUE has the significance of an upper limit, while if the INCREMENT is negative (i.e., a decrement) then the FINAL VALUE has the significance of being a lower limit for the LOOP COUNTER.

The statement that follows the word "do" is repeatedly executed once for each value assigned to the LOOP COUNTER. Of course, after each assignment of a value to the LOOP COUNTER, the test is made to see if its value remains within its proper range. If not, the repetition is discontinued, and execution of the for statement is complete. We then go on to execute the next ALGOL statement.

Here are some examples of the for clause and its equivalent iteration box. Be sure to study each example carefully--especially Example 4.

Example	for clause	Iteration box

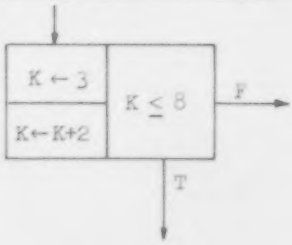
1. for I := 1 step 2 until N do



Means: Execute the statement that immediately follows the word do once for each value of I in steps of 2 until I ≤ N is false (or, if you like, until I > N is true). Thus, if N has a value of 7, the statement would be repeated (four times) for values of I of 1, 3, 5, and 7. When the counter is incremented again, its value exceeds 7 and we exit from the loop by proceeding to execute the next statement after the for statement.

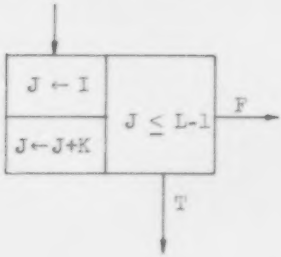
Example	for clause	Iteration box

2. for K := 3 step 2 until 8 do



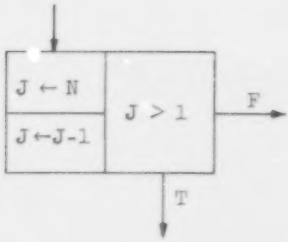
Means: Execute the statement immediately following the word "do" once for each value of K until K ≤ 8 is false. K is given an initial value of 3 and is incremented each time by 2. The statement is executed for K = 3, 5, and 7 (three times).

3. for J := I step K until L - 1 do



Means: Execute the statement immediately following the word "do" once for each value of J until J ≤ L - 1 is false. J is given a starting value equal to that currently assigned to I. The increment is the current value of K.

4. for J := N step -1 until 2 do



Means: Execute the statement immediately following the word "do" once for each value of J as it descends in value from N in steps of -1 until it is no longer greater than or equal to 1(i.e., N - 1 times).

We might now take a look at one more application of the for statement and then do some exercises. Figure A4-3 shows how we would write the ALGOL program for the flow chart you drew in answering Problem 1 of Exercises 4-1 in the flow chart text.

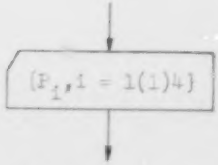
ALGOL

```
begin integer I, ID;
      real A, B, C, D;
      for I := 1 step 1 until 50 do
      begin
            read (ID, A, B, C);
            D←sqrt(A↑2 + B↑2 + C↑2);
            print (ID, A, B, C, D);
      end;
end
```

the whole for statement

Figure A4-3. Another illustration of the for statement

In an exercise where you are interested in writing the ALGOL equivalent of a flow chart box like:



you should consider the three alternatives that are now available to you.

1.
```
read (P[1], P[2], P[3], P[4]);
```

2.
```
      I := 1;
BOXA: read (P[I]);
      I := I + 1;
      if  I < 4  then go to BOXA;
```

3.
```
for  I := 1 step 1 until 4 do read (P[I]);
```
a for statement

Pick whichever form pleases you. They will all do the job. However,

there is one aspect to the alternatives which should be pointed out. The first method

```
read(P[1], P[2], P[3], P[4]);
```

will activate the reading mechanism and continue until four numbers have been read. The data themselves may all be on one card or two to a card, or each on a separate card.

The second and the third methods, on the other hand, activate the reading mechanism anew each time to read one number, P[I]. Data for these two forms must be punched one to a card since only one number is read from each card.

Burroughs (Extended) Algol allows an interesting variation, a for clause inside the read statement. The following statement is legal in Extended Algol:

```
READ (FOR I ← 1 STEP 1 UNTIL N DO P[I]);
```

This statement will read, one or more cards, until N numbers have been read, assigning them sequentially to the $(P_i, i = 1(1)N)$.

Exercises A4-1

1 - 3. For each of the other flow charts you drew in answering Problems 2, 3, and 4 of Exercises 4-1, now write an equivalent ALGOL program.

4. For Problem 5 of Exercises 4-1 write an ALGOL program assuming the number of employees never exceeds 100.

A4-2 Illustrative examples

There are a number of simple examples of loops in Section 4-2 of your flow chart text which can be easily translated into ALGOL code with the aid of the for statement. We shall use these to further illustrate some of the details in the proper use of the for statement. Figure A4-4 shows ALGOL coding equivalent to the flow charts in Figure 4-8.

```
for  I := 1 step 1 until N do
     begin
          Y := X[I] ↑ 3;
          write (X[I], Y);
     end;
```

(a)

```
for  J := 1 step 1 until N do
     begin
          read (X);
          Y := X ↑ 3;
          write (X,Y);
     end;
```

(b)

Figure A4-4. ALGOL for flow charts in Figure 4-8

Similarly, we see in Figure A4-5 an ALGOL equivalent of the flow chart in Figure 4-9(a).

```
SUM := 0;
for  J := 1 step 1 until N do
SUM := A[J] + SUM;
write (SUM);
```

Figure A4-5. ALGOL for Figure 4-9(a)

Before looking at our next translation, try your hand at writing the ALGOL for Figure 4-9(b). Now compare your code with that in Figure A4-6.

a single for statement

```
MAX := abs(A[1]);
for  J := 2 step 1 until N do
if abs A[J] > MAX then MAX := abs A[J];
write (MAX);
```

Figure A4-6. ALGOL for Figure 4-9(b)

Notice that the statement which is to be repeated, under control of loop counter J, is an if statement. The semicolon after A[J] marks the end of the for statement.

In summary, we see that any single if statement, or read, or write, or assignment statement may be the candidate for repetition. If more than one statement is to be repeated, we "weld" them into one by forming one compound statement with the aid of begin and end.

Exercises A4-2 Set A

In each of the following exercises we present ALGOL code for the flow charts in Figure 4-10(a), 4-10(b), 4-11(a) and 4-11(b), respectively. Your job is to indicate what errors, if any, have been made in the coding process. The necessary declarations are to be disregarded here.

1. For Figure 4-10(a):

```
MAX := abs (A1);
INDEX := 1;
for  J := 2 in steps of 1 until N
if abs(A[J]) > MAX then
     begin  MAX := abs(A[J]);
            INDEX := J;
     end;
write (INDEX, MAX);
```

2. For Figure 4-10(b):

```
MAX := abs A([2])
for  J := 4 step 2 until N do
     if abs(A[J]) > MAX; then MAX := A[J];
write (MAX);
```

3. For Figure 4-11(a):

```
FACT := 1;
for  k := 1 step 1 until N do;
     FACT := K × FACT;
     write (K, FACT);
```

4. For Figure 4-11(b):

```
begin LTERM := 1;
      NLT := 0; end;
      for  k := 1 step 1 until N do
      begin  COPY := LTERM;
             LTERM := LTERM + NLT;
             NLT := COPY;
      end;
      write (K, LTERM);
```

Exercises A4-2 Set B

1-17. For each of the flow charts you drew in answering the exercises of Section 4-2, Set A, now write the equivalent ALGOL statements as partial programs only. Do not bother to write the needed declarations.

For Problem 12: Is the following code correct? If not, why not?

```
for  I := 1 step 1 until N do
if  abs(P[I]) > 50 then begin
      W := P[I]; ANY := 1; end;
ANY := 0;
```

You may now be interested in seeing how the flow chart of the factors-of-N algorithm (Figure 4-14) is coded in ALGOL. Here it is in Figure A4-7. If you have done the exercises in the preceding set, you should have no problem in following the program in this figure.

```
begin comment Finding the factors of N;
      real BOUND;
      integer N, K, L;
      read (N);
      BOUND := sqrt(N);
      write ("The factors of", N, "are");
      for  K := 1 step 1 until BOUND do
      if  N = K × entier(N/K) then
            begin L := N/K;
                  write (K,L); end;
end
```

Note: this is one for statement

Figure A4-7. ALGOL for factors of N equivalent to Figure 4-14.

Similarly, you should have no problem following the way we code the polynomial evaluation algorithm (Figure 4-17) in ALGOL. This is shown in Figure A4-8.

```
begin       comment Polynomial evaluator. Sunday method;
            real array B[0:3];
            real  X, VALUE;
            integer I, K;
BOX1:       for  I := 0 step 1 until 3 do read (B[I]);
            read (X);
            VALUE := B[0];
            for  K := 1 step 1 until 3 do
            VALUE := VALUE × X + B[K];
            write ("The value is", VALUE);
end
```

Figure A4-8. ALGOL equivalent of the polynomial evaluation (Figure 4-17)

In the statement labeled BOX1, the use of a for clause allows us to repeat the statement:

```
read B[I];
```

for various values of I. This is the equivalent of the flow chart notation

$$\{B_I,\ I = 0(1)3\}$$

Exercises A4-2 Set C

For each of the flow charts you drew in answering the three problems of Exercises 4-2, Set B, now write the equivalent full ALGOL programs. For Problems 1 and 2 assume that N will never exceed 50. Check the Main Text for details on Problem 3(b).

A4-3 Table-look-up

We now test our ability to write the ALGOL equivalent to the table-look-up (Figure 4-24) using the bisection method. Figure A4-9 shows the program. We assume for the ALGOL program that the table to be stored will never contain more than 200 X's and 200 Y's.

```
begin       comment  Table-look-up by bisection;
            real array  X[1:200], Y[1:200];
            real A;
            integer  N, K, LOW, HIGH, MID;
            read (N);
            for  K := 1 step 1 until N do read (X[K], Y[K]);
            read (A);
            if  X[1] ≤ A then begin
                if  A ≤ X[N] then go to BOX5; end;
            write (A, "is not in the range of the table.");
            go to HALT;
BOX5:       LOW := 1;
            HIGH := N;
BOX6:       if HIGH-LOW = 1 then begin
            write (X[LOW], Y[LOW], A, X[HIGH], Y[HIGH]);
            go to HALT; end;
            MID := entier((LOW + HIGH) / 2);
            if  A ≤ X[MID] then
                begin HIGH := MID;
                      go to BOX6; end
                else begin  LOW := MID;
                      go to BOX6; end;
HALT:
end
```

Figure A4-9. ALGOL equivalent of table-look-up by bisection

Study the correspondence between this ALGOL program and its flow chart. By now you should realize there are many equivalent ALGOL programs. The one in Figure A4-9 is only one of several possible ones. You may prefer to code the ALGOL somewhat differently.

Exercises A

Write an ANSI COBOL program equivalent to the flow chart in Figure A-3-5. Assume that we will not need to store more than 200 X,Y pairs in memory at any one time.

A4-4 Nested loops

Just as one loop, with its iteration box, can form part of the computation portion of another loop, we can have one complete for statement become part of another. The statement which immediately follows a for clause (i.e., right after the word do) can either be a for statement itself or it can be a compound statement one of whose parts is a for statement.

For our first example, examine Figure 4-29, beginning with Box 4. We now show the equivalent ALGOL code in Figure A4-10.

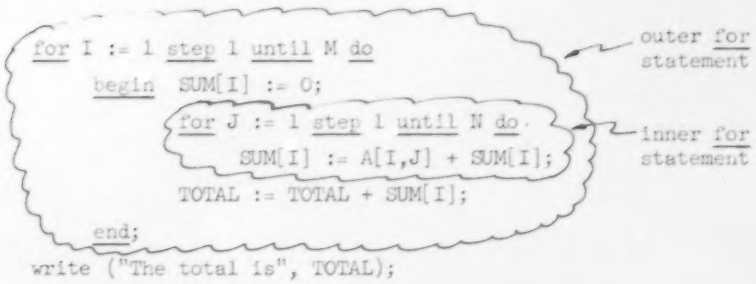

for I := 1 step 1 until M do
begin SUM[I] := 0;
for J := 1 step 1 until N do
SUM[I] := A[I,J] + SUM[I];
TOTAL := TOTAL + SUM[I];
end;
write ("The total is", TOTAL);
outer for statement
inner for statement

Figure A4-10. Nested for statements

The indentation of the statements helps to suggest the idea of nesting. The compound statement which follows the word do of the first for clause is the computation portion of the outer loop. There are three parts to this compound statement, the second one being another for statement which has its own computation portion. When executing these ALGOL statements the computer can keep track of what loop it is in at all times.

Now let's see how to write the whole program for Figure 4-29. We assume the matrix can have up to 50 rows and 50 columns. Data for the matrix entries will be assumed to be punched in row by row order, one number per data card.

Our first problem is to express Box 2 in ALGOL. You may remember that we sidestepped a problem like this in Section A3-6 and promised we would tell you how to do this in Chapter A4. You have probably already figured how to do this with nested for statements as shown in Figure A4-11.

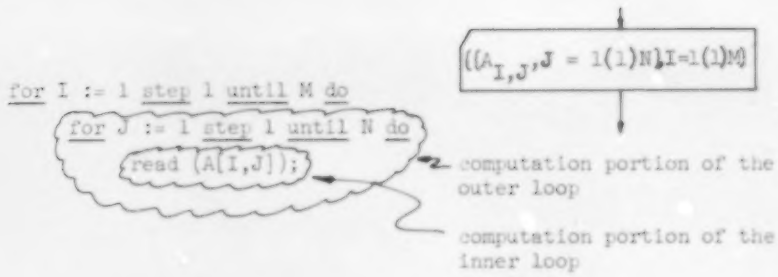

((A I,J ,J = 1(1)N),I=1(1)M)
for I := 1 step 1 until M do
for J := 1 step 1 until N do
read (A[I,J]);
computation portion of the outer loop
computation portion of the inner loop

Figure A4-11. Equivalence between box 2 of flow chart and nested for statements

After studying this figure you can probably see how we might print out the matrix entries in row-by-row order. We could say:

```
for I := 1 step 1 until M do
    for J := 1 step 1 until N do
        write (A[I,J]);
```

How would you print out the entries in column-by-column order? Just reverse the order of the two for clauses:

```
for J := 1 step 1 until N do
    for I := 1 step 1 until M do
        write (A[I,J]);
```

We are now ready to write the whole ALGOL program for Figure 4-29. Here it is in Figure A4-12.

```
begin    comment summing entries of a matrix;
         real array  A[1:50, 1:50], SUM[1:50];
         comment  We need to reserve 50 locations for the SUM vector
              and enough locations for a 50 X 50 matrix  A;
         real TOTAL;
         integer I, J, M, N;
         read (M,N);
         for I := 1 step 1 until M do
              for J := 1 step 1 until N do
                   read (A[I,J]);
         TOTAL := 0;
         for I := 1 step 1 until M do
              begin SUM[I] := 0;
                   for J := 1 step 1 until N do
                        SUM[I] := A[I,J] + SUM[I];
                   TOTAL := TOTAL + SUM[I];
              end;
         write ("The total is", TOTAL);
end
```

Figure A4-12. ALGOL equivalent of Figure 4-29.

Exercises A4-4 Set A

1-8. For each of the flow charts you constructed for the exercises in Section 4-4, Set A, write the equivalent ALGOL statements. Don't bother writing declarations unless you feel they add to your understanding of the translation problem.

Triply-nested loops are just as easy to write in ALGOL by the nesting of for statements as they are to draw in the flow charts. Figure A4-13 shows how the "stickler" in Figure 4-31 would be coded in ALGOL.

```
begin    comment  The Stickler;
         integer H, T, U;
         for H := 1 step 1 until 9 do
              for T := 0 step 1 until 9 do
                   for U := 0 step 1 until 9 do
                        if 100 × H + 10 × T + U = H ↑ 3 + T ↑ 3 + U ↑ 3
                             then write (H, T, U);
end
```

Figure A4-13. The stickler in ALGOL

Exercises A4-4 Set B

Write ALGOL programs for the flow chart solutions you obtained for Problem 7, Exercises 4-4 Set B in the Main Text.

Exercises A4-4 Set C

1. Write a complete program for the Prime Factorization Algorithm, Figure 4-32.
2. Write a complete ALGOL program for the shuttle interchange sorting algorithm, Figure 4-34. Assume you may wish to sort up to 500 numbers.
3. Write a complete ALGOL program for the sort algorithm shown in Figure 4-35. Make the same assumption in this program that you are asked to make in the preceding exercise.
4. Write a complete ALGOL program for finding the longest decreasing subsequence. Base your program on the flow charts in Figures 4-38 and 4-39. Assume the given sequence will not exceed 100 values in all.

ALGOL Chapter A5

PROCEDURES

A5-1 Procedures

The ALGOL translation of a reference flow chart is called an ALGOL PROCEDURE. This procedure looks very much like a complete ALGOL program except within the initial group of assertions, corresponding to the flow chart funnel. The head of a procedure consists of an introductory symbol (to be explained shortly), the name of the procedure, a list of parameters and something about their types. This head is followed by a body to carry out the task of the procedure.

The entire ALGOL procedure (the head and the body) is to be thought of as a declaration. This procedure declaration must be included in the head (along with such declarations as real, integer, etc.) of any program using the procedure.

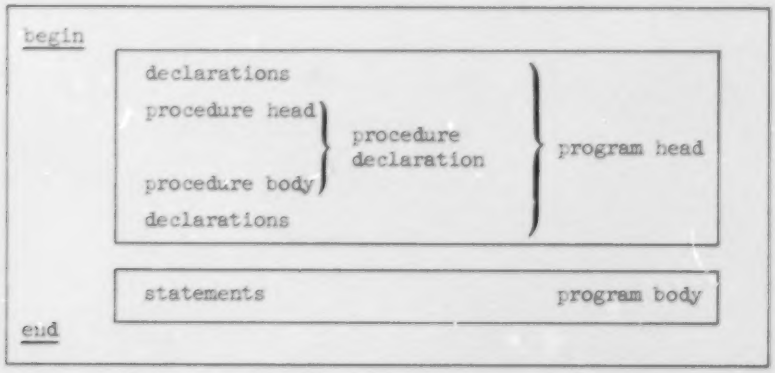


Figure A5-1. Structure of an ALGOL Program including a procedure

Earlier in Chapter 2 we defined an ALGOL program as a sequence of declarations followed by a sequence of statements, with the whole thing enclosed by "begin" and "end". Another name for a program is a block. A block must include at least one declaration after the begin. Otherwise, we may have a compound statement rather than a block.

The body of a procedure declaration may be a single statement. For example:

```
integer procedure parity(i);                    } procedure head   } procedure
        integer i;                                                   declaration
if i = 2 × entier(i/2) then parity := 0 } procedure
else parity := 1;                         } body
```

The body of a procedure may also be a compound statement. If local variables are required, the body will be a whole program or block.

An ALGOL procedure to evaluate a function and report a single value is called a function procedure. (A second type of ALGOL procedure will be encountered in Section A5-4.) For a function procedure, the flow chart funnel corresponds to an assertion at the beginning of the procedure declaration. This assertion begins with the words real procedure if the value to be reported is real, or integer procedure if the value to be reported is an integer. These words are followed by the name of the function, its argument in parentheses and a semicolon. For example:

```
real procedure sqroot(y);
```

The head of this procedure declaration concludes with the type declaration for y. The use of this function procedure is shown in more detail in Figure A5-2.

```
begin comment  The head of the program using the sqroot procedure comes
               first.  It is made up of the usual type declarations and
               the procedure declaration for sqroot.;
    real       y, x, n, z;
    ----
    ----
    comment    The procedure declaration for sqroot follows;
    real procedure sqroot(y);
        real y;
        begin
            ----
            sqroot := ----
        end
    comment    The body of the program using the sqroot
               procedure completes this example;
               ----
               ----
               z := a + sqroot(x);
               ----
               ----
               if  z = 0 then z := a - sqroot(n);
               ----
end
```

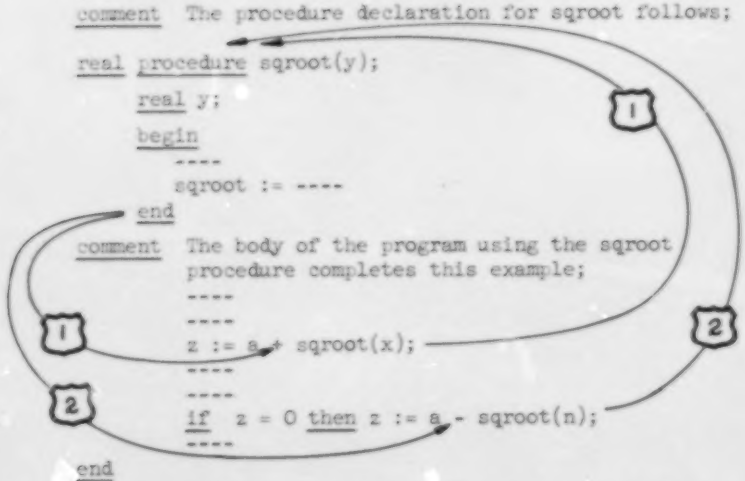


Figure A5-2. Use of an ALGOL function procedure

Since special symbols like $\sqrt{\ }$ are not available in ALGOL, we replace such a symbol with an alphabetic name for the function (in this case, sqroot). Any name can be chosen except for unusual reserved words.

You are already familiar with the use of standard mathematical functions like sqrt, sin, abs, etc. (See Table A2-2). ALGOL procedures are different from these standard mathematical functions, even though our first example, sqroot, serves the same purpose as sqrt. The difference is not just in the way the names are spelled but is mainly in the fact that the standard mathematical functions are part of the compiler system. The techniques that might be used for adding to the list of standard mathematical functions are outside of the scope of this book. Function procedures provide a way that you can develop whatever set of reference programs you want.

The body of an ALGOL procedure may be enclosed by a begin and an end. Just as reference flow charts sooner or later reach a return box, an ALGOL procedure may eventually reach a final enclosing end. This end corresponds to the flow chart return box shown in Figure 5-5. Since the end in an ALGOL function procedure does not indicate what variable is to be reported to the main program, a convention is needed to identify the value to be reported. The convention used is that the name of the function procedure itself must appear at least once on the left side of an assignment statement, and its value, when the procedure is completed, is the value reported. This convention must be observed for all function procedures whether the body is enclosed in begin-end or not.

In other respects, an ALGOL procedure must conform to the requirements of any ALGOL program. In particular, the type of all variables used in the procedure must be declared.

An ALGOL procedure is a self-contained unit one expects to use many times. It is especially convenient to compile procedures separately, rather than along with the programs in which they are going to be used. Then one can develop a library of procedures which can be used at any later time. Some ALGOL compilers allow for the separate compilation of procedures; others do not. Find out which applies for the compiler you are using. In this chapter we will follow the original definition of ALGOL, which, without forbidding it, does not tell how to achieve separate compilation of procedures.

Figure A5-2 is intended to relate to Figure 5-6 of the flow chart text illustrating features related to ALGOL procedure use. The first time the ALGOL procedure is required by the "function designator", sqroot(x), we go to the real procedure declaration via route 1. This declaration directs that the value of x be assigned to y in the procedure. It is essential that the variable in the function designator agree in type with the parameter declared in the procedure declaration. That is, since y has been declared real in the procedure declaration, declaration of x to be integer in the calling program would be in error.

When the execution of the procedure has been completed, a value has been assigned to sqroot and the return to the statement that requested the procedure is by route 1. Upon return to the ALGOL program, the value assigned to sqroot is added to a and the result assigned to z. At the next function designator (in the conditional statement) we go to the real procedure declaration by route 2, substitute n for y in the procedure (note that they are both real), execute the procedure and return via route 2 to the conditional statement, the result of the procedure having been assigned to sqroot.

An actual function procedure declaration for the square root can be prepared with reference to Figure 5-7 and is shown in Figure A5-3.

```
comment square root function procedure;
real procedure sqroot(y);
    real y;
       begin real g, h;
             g := 1.0;
       BOX2: h := 0.5 × (g + y/g);
             if abs(h - g) < 0.0001 then go to BOX5;
             g := h;
             go to BOX2;
       BOX5: sqroot := h;
       end
```

Figure A5-3. A function procedure declaration for square roots

As you inspect this function procedure, we take the opportunity to warn you about something you are unlikely to do anyway. That is, do not assign to an argument (in this case y) of a function procedure! It is never necessary to do this and with some ALGOL compilers it can produce very dire results. You will be reminded of this danger area once more in Section A5-5.

Exercises A5-1

1 - 3. Write ALGOL function procedures for the flow charts prepared in Exercises 5-1, main text.

A5-2 Functions and ALGOL

The flow chart text tells us that we can view any flow chart which, given a value, will produce another value as the evaluation of some function. The statement is as true for ALGOL programs as for flow charts. Although mathematical functions exist which cannot be evaluated, either with a flow chart or by an ALGOL program, the common usage of the word function in computing is strictly limited to those which can be evaluated with a flow chart. In computing, then, a function is commonly thought of as a relationship for which a reference flow chart is used.

The domain of a function, in computing, is the set of values that the argument in the funnel of the flow chart can take on. The range of a function is the set of values that can be reported to the main flow chart. In ALGOL, the domain can be either a set of real numbers representable in computer or a set of integers representable in a computer.

A5-3 ALGOL function procedures

ALGOL function procedures can have as many arguments as are necessary. The min function provides an example . The procedure declaration in Figure A5-4 corresponds to Figure 5-14 of the flow chart text.

```
comment  minimum of two arguments function procedure;
real procedure  min (b, c);
     real b, c;
       begin  real z;
              if b > c then go to BOX3;
              z := b; go to BOX4;
              BOX3:  z := c;
              BOX4:  min := z;
       end
```

Figure A5-4. A function procedure of two arguments

Notice that the type of each variable has been declared to be real by the three type declarations. That is, the function procedure expects to receive two real values and to report a real value. The variable z is also declared to be real. The procedure declaration could be changed easily to expect integer values or to report an integer value by appropriately substituting integer for real. In Figure A5-4 we do not need to introduce the variable z since min can serve the same role. For this reason Figure A5-4 can be replaced by Figure A5-5.

```
comment  minimum of two arguments function procedure;
real procedure  min (b, c);
     real  b, c;

              if b > c then min := c else min := b;
```

Figure A5-5. Improved min procedure declaration

Figure A5-5 contains other improvements. Not only has the extraneous variable, z, been dropped , but assignments have been included in the if-clause. The body of the procedure declaration is then just one statement.

The parameter list of an ALGOL procedure declaration may contain integer or real variables, variables containing alphanumeric information, names of vectors and names of arrays. In every case there must be a one-to-one correspondence in both number and type between parameters in the (actual) list of the function designator and those in the (formal) list of the procedure declaration. Confusion would reign if we tried to refer to the procedure declaration of Figure A5-5 by writing something like

t = min(a, b, c)

or

t = min(m, p) where m has been declared to be an integer.

One function procedure declaration may use another function procedure. Both declarations can then appear in the head of the main program. See Exercises A5-3, Set B.

A classification of variables

The distinction between local and nonlocal variables with respect to ALGOL procedures is the same as is described in the flow chart text. Thus, in the two procedures for min given in Section A5-4, the arguments b and c are nonlocal variables. In the first procedure z is a local variable.

Difficulties can arise if nonlocal variables are changed within a function procedure declaration. What can happen depends to some extent on the particular compiler that is being used. Unless you are absolutely sure about what can happen, avoid changing nonlocal variables in function procedures!

An appreciation of the distinction between local and nonlocal variables can be had by recalling that procedures are separate units. Upon leaving a procedure, the values of local variables are lost; the storage space used to keep this information is freed. Nonlocal variables, on the other hand, remain defined after leaving a procedure.

A local variable in an ALGOL procedure is completely independent of any variable with the same name that might appear outside the procedure. Notice that the local variable is declared in the body of the procedure, and not along with the specification of the variables which appear in the parameter list. We will always insist that nonlocal variables be given in the parameter list. There is another way in which they can appear in ALGOL procedures, but we will not consider this other possibility.

Composition of function designators

The situation with respect to composition of function designators is exactly as described in the flow chart text. This is just what you have learned as composition of functions. That is, given function procedure declarations defining f1(x) and f2(x), one can write

```
y := f1(f2(x));
```

so long as the range of f2 is a subset of the domain of f1. Correspondingly, the following function designators are entirely proper:

```
y := min(abs(A+B),5.4);
or  y := min(min(f,abs(t)),Q);
or  y := min(sqroot(b × b - 4.0 × a × c), -6);
or  y := sqroot(min(x,y));
```

Exercises A5-3 Set A

1 - 6 and 7(a). Write ALGOL programs or function procedures corresponding to the flow charts prepared in Exercises 5-3, Set A, main text.

Exercises A5-3 Set B

1. Write an ALGOL function procedure for the flow chart of the Euclidean Algorithm you prepared in Problem 1 of Exercises 5-3, Set B, main text.
2. Write an ALGOL function procedure for Problem 2 of Exercises 5-3, Set B, the greatest common factor of three integers.
3. Write an ALGOL program that corresponds to the flow chart for determining
   (a) the number of non-similar triangles;
   (b) the sum of the perimeters of the non-similar triangles corresponding to the flow charts you prepared in Problem 3 of Exercises 5-3, Set B.
4. Write an ALGOL program for the algorithm of Problem 4, Exercises 5-3, Set B. Try to estimate how much computation will be involved. Measure computation in terms of the number of additions, subtractions and comparisons that must be made, counting each as 1.

A5-4 ALGOL "proper" procedures

ALGOL procedures which correspond to reference flow charts for procedures that are not functions are called "proper" procedures. Corresponding to the funnel of the reference flow chart is an assertion beginning:

```
procedure sort(n,d);
    integer n;
    real array d;
```

The word procedure is followed by the name of the procedure being declared, a parameter list in parentheses and a semi-colon. In the sort example we see that an entire vector is identified by its name d in the parameter list. No attempt is made to subscript d in the parameter list but d is declared as a real array. Of course, n is declared as an integer.

Since a proper procedure does not report a value in the same way that a function procedure does, the name of the procedure will not appear in the body of the procedure and we do not attach real or integer to the declaration defining the procedure identifier.

A proper procedure declaration corresponding to Figure 5-16 is given in Figure A5-6. The dimension of the array d does not need to be specified in the procedure declaration. Whatever the dimension of d may be (and it must be specified in the ALGOL calling program), n is the actual number of components of d to be sorted. Thus, the procedure declaration is usable for a vector with any number of components.

```
comment proper procedure for sort;
    procedure sort(n,d);
        integer n;
        real array d;
        begin
            integer i,j;
            real b;
            for i:=1 step 1 until n - 1 do
                for j:=i+1 step 1 until n do
                    if d[i] > d[j] then
                        begin
                            b:=d[j]; d[j]:=d[i]; d[i]:=b;
                        end;
        end
```

Figure A5-6. Procedure declaration for sorting

Look at Figures 5-16 and A5-6 side by side. See how the ALGOL statements correspond to the flow chart boxes. Notice also that values can be intentionally assigned to parameters of a proper procedure. We warned you not to do this in function procedures but it is right and proper here. This is how a proper procedure produces its output.

Use of a proper procedure is activated by a "procedure statement", analogous to the execute box of the flow chart text. The procedure statement consists solely of the procedure name followed by the parameter list in parentheses and a semicolon, for example:

```
sort(88,b);
```

where b is an array having at least 88 components in the calling ALGOL program. In ALGOL the process of referring to a procedure is termed "calling the procedure".

An ALGOL program calling sort, corresponding to Figure 5-23, is shown in Figure A5-7. This program assumes that the maximum dimension of b and c

```
begin
     comment a program to illustrate procedure statements;
     real array  b[1:100], c[1:100];
     integer  k,i;
     comment the procedure declaration for sort (Figure A5-6) must
             be inserted here;
     read  (k);
     for  i:=1  step 1 until k do read (b[i]);
     for  i:=1  step 1 until k do read (c[i]);
     sort  (k,b);
     sort  (k,c);
     for  i:=1  step 1 until k do
          begin
               write  (b[i]);
               write  (c[i]);
          end;
end
```

Figure A5-7. A program calling the sort procedure

is 100. We also call attention to the comment in the head of the program which says that the procedure declaration for sort must be inserted. This means that Figure A5-6 is to be physically placed in the head of the program. We make this comment rather than the actual insertion in the calling program here so that the figures can be more easily scanned by the eye.

Exercises A5-4 Set A

1 - 5. For the flow charts prepared in Exercises 5-4, Set A, main text, write ALGOL programs and procedures.

Exercises A5-4 Set B

1 - 3. For the flow charts prepared in Exercises 5-4, Set B, main text, for Problems 1, 2 and 3, write ALGOL programs and procedures.

Exercises A5-4 Set C

1 - 4. For the flow charts prepared in Exercises 5-4, Set C, main text, Problems 1 through 4, write ALGOL programs and/or procedures.

A5-5 Alternate exits and techniques for branching

Provisions for alternate exits and branching from ALGOL proper procedures can mirror the initial discussion in the flow chart text. A parameter is provided to indicate the results of tests performed by the procedure. Figures A5-8 and A5-9 present a procedure declaration and a calling program corresponding to the flow charts of Figure 5-26 and Figure 5-27.

```
comment  a procedure to test equality of two complex numbers;
     procedure  compeq(a,b,c,d,n);
     real  a, b, c, d;
     integer  n;
     begin
          if  a = c  then begin if  b = d  then  n:=0  else
                     n:=1  end else  n:=1;
     end
```

Figure A5-8. A procedure declaration for compeq

```
begin
     comment  pieces of a program showing the use of  compeq;
     real  x, y, u, v;
     integer  k;
     comment  the procedure declaration of Figure A5-8 must be
              physically inserted here;
          ---
          ---
          compeq(x,y,u,v,k);
          if  k = 0  then go to  st4   else go to  st3;
st4:      ---
st3:      ---
end
```

Figure A5-9. Pieces of an ALGOL program testing the equality of two complex numbers

Function names and statement labels as procedure parameters

In ALGOL both function names and labels are allowed as parameters of procedures. In the head of the procedure declaration these parameters must be specified as to type. For example:

```
procedure SMSG(x,y,f,e);
real x,y;
real procedure f;
label e;
```

The actual parameters which replace the formal ones in the call of this procedure must match the types specified. For instance, if the call is

```
SMSG(R1,R2,R3,R4);
```

then R1 and R2 must have been declared to be real in the head of the calling program. R3 must have been declared as a real procedure (function) and R4 must be a label. ALGOL 60 does not require labels to be declared in the calling program. (Extended ALGOL does require this.)

Many procedures can be written more naturally and easily in ALGOL by including functions or labels as parameters. In Chapter 7 it would be difficult to write some of the procedures in any other way.

Exercises A5-5

1. Following the flow chart prepared in Problem 1, Exercises 5-5, main text, write an ALGOL procedure to solve two equations in two unknowns.

2. (a) - (d). Write the ALGOL corresponding to parts (a) and (b) or (c) and (d) of Problem 2 of Exercises 5-5. Use either complete program on the following quadratic equations:

   (e) $2x^2 - 3x + 7 = 0$

   (f) $3.14x^2 - 6.2x - 14.23 = 0$

3. Write an ALGOL procedure and the calling program corresponding to one of the techniques you used for solving Problem 3, Exercises 5-5, main text.

†A5-6 Symbol manipulation in ALGOL

In Section A2-8 the input and output of alphanumeric characters was discussed. Now we want to find out how alphanumeric data can be processed so that we will be able to alter such input data as

THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG

or

3.14159

or

$$r + s(t + u(v + w)).$$

Since we will want to be able to refer to each individual element in such character strings, we will associate a separate variable with each element of a string.

In Section A2-8 you were told about the read symbol statement to input alphanumeric data. You may remember that this statement appeared as, for example,

readsymbol(a,b,c);

which would read whatever symbols were punched in the first three columns of a card and assign these symbols to a, b and c, respectively, before proceeding to the next statement.

Comment: In Burroughs (or Extended) ALGOL, symbol manipulation is especially easy. Variables whose values are alphabetic, numeric or other characters, are declared as a separate type called alpha.

Example: alpha a,b,c;

Each such variable can store up to six alphanumeric characters. ALPHA arrays can also be declared. It is possible to read or write characters with a free-field read; however, it is often more convenient to store one ALPHA character per ALPHA variable. In Extended ALGOL this can be done with a read statement like the following:

read(< 72A1 >, for I ← 1 step 1 until 72 do C[I]);

where 72A1 is a format of 72 one-character ALPHA types. This statement will read 72 columns of alphanumeric information. A corresponding write statement can be formed.

A5

In ALGOL 60, can the readsymbol statement be used to read whole strings? If the string we want to read is punched on a card, and the punching goes all the way across the card (as you would naturally do in punching a long English sentence such as this one), readsymbol would have to be followed by a parameter list with eighty identifiers (one for each card column). Iteration on the readsymbol statement doesn't help reduce the number of identifiers needed since a new card is read for each readsymbol execution.

The way out of this is indicated by noting that the readsymbol statement looks just like a sort statement or any other procedure statement. In fact, that is almost what it is! The statements read, print, readsymbol and printsymbol are all similar to procedures. They are written in machine language and included in the various ALGOL compilers which automatically insert them when needed. None is a basic part of ALGOL and that is why input-output procedures may vary in name and in the details of their specifications from one implementation to another.

All we need to do to be able to read and print strings is to define new procedures. Let us imagine that symbols are to be input from a string of indefinite length (like symbols on a punched paper tape or on cards laid end to end). Now we can define a procedure (and leave it to an expert to prepare it) with the declaration head:

```
procedure  getsymbol(s,n);
integer  n;
integer array  s;
```

The purpose of getsymbol is simply to read the next symbol of the input string and assign it to the nth element of s. The procedure will read a new card whenever it is needed to be sure that a next symbol is always available.

Now we define a procedure declared by

```
comment  procedure declaration for reading strings;
procedure  readstring(length,string);
integer  length;
integer array  string;
begin integer  i;
    for  i:=1  step  1  until length do
        getsymbol(string,i)
end
```

We can also define complimentary procedures called putsymbol and printstring to provide for the output of strings.

We are now ready to write a proper procedure, corresponding to Figure 5-28, for chekch. This declaration is shown in Figure A5-12.

```
comment  a procedure declaration to search for a character;
procedure  chekch(n,s,m,c,p);
     integer  n, m, c, p;
     integer array  s;
     begin
          integer  i;
          for  i:=m  step 1 until n do
               if s[i] = c then
                    begin p := i;
                         go to endchekch;
                    end;
          p:=0;
     endchekch:
     end
```

Figure A5-12. A character searching procedure

One of the things to notice about the procedure declaration in Figure A5-12 is that no limitation is made on the length of the string. The same declaration can be used to inspect a string of length 5 or a string of length 5000 (if enough memory is available). Another thing to notice is that two characters can be compared by the relation " = ".

An ALGOL procedure corresponding to Figure 5-34 is given in Figure A5-13.

```
comment  procedure declaration for  chekst;
procedure  chekst(n,s,m,k,c,p);
     integer  n, m, k, p;
     integer array  s, c;
     begin
          integer  ℓ, r, j;
          comment  the procedure declaration for  chekch  must
                   be inserted here;
          ℓ:=m;
repeat:   if  ℓ > n - k + 1  then go to zero;
          chekch(n,s,ℓ,c[1],p);
          if  p = 0  then go to  endchekst;
          if  p > n - k + 1  then go to zero;
          r:=p+1;  j:=2;
test:     if  j > k  then go to  endchekst;
          if  s[r] = c[j]  then
               begin ℓ := p + 1;
                    go to repeat;
               end;
     end;
```

Figure A5-13. A procedure inside a procedure

One way to code the procedure declaration for chekst is seen in Figure A5-13. We know that there are various forms to code a flow chart. Each form may be an equally valid program; the choice of one over another is often a matter of taste. To illustrate this point we present Figure A5-13a which is equivalent to Figure A5-13 but has a very different appearance. Which do you prefer?

```
procedure  chekst(n,s,m,k,c,p);
integer  n, m, k, p;
integer array  s, c;
begin
    integer  i, r, j;
    comment  the procedure declaration for  chekch  must
             be inserted here;
    i:=m;
repeat:  if  i ≤ n - k + 1  then
         begin
             chekch(n,s,i,c[1],p);
             if  p ≠ 0  then
             begin
                 if  p ≤ n - k + 1  then
                 begin
                     r:=p+1;
                     for  j:=2 step 1 until k do
                     begin
                         if s[r] = c[j] then
                             r:=r+1 else
                             begin i:=p+1;
                                 go to repeat;
                             end;
                     end;
                 end
                 else  p:=0;
             end;
         end
    else  p:=0;
end
```

Figure A5-13a. An alternate procedure declaration for chekst

The interesting thing about either of these procedure declarations is that they make use of another procedure and so contain a "subprocedure declaration". An inspection of the structure of Figure A5-13 shows us how procedures can use other procedures to any level. Figure A5-14 displays this structure.

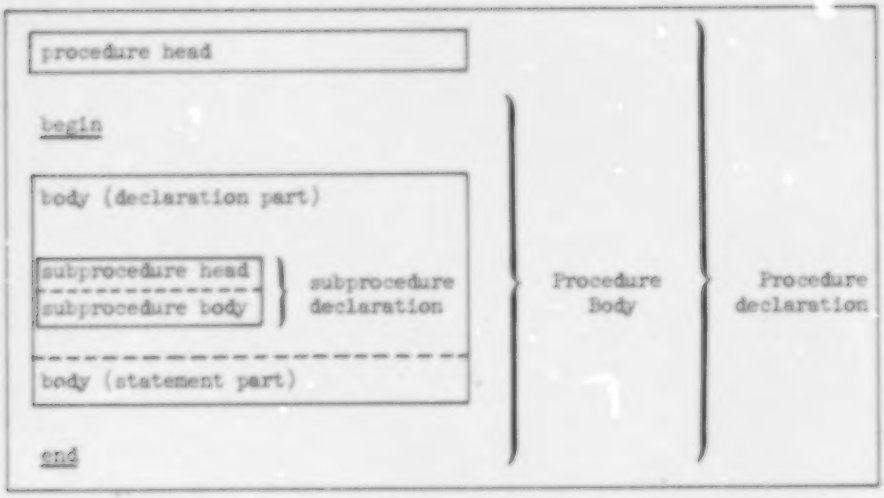


Figure A5-14. Structure of nested procedure declarations

One further point should be made here. It is often very convenient to consider the length of a string, and the string itself, as being part of a single entity. One way to do this is to use an array, say str, with the property that its first component str[0] is an integer equal to the length of the string, while its remaining components str[1], str[2], and so on, are the characters themselves.

If we now denote the character for which the search of chekch is being made by char, the procedure declaration for chekch is shown in Figure A5-15.

```
comment  modified procedure declaration for chekch;
procedure  chekch(str,m,char,p);
     integer  m, char, p;
     integer array  str;
     begin
          integer  i;
          for  i:=m   step 1 until str[i]   do
               begin
                    if  str[i] = char then begin p:=i;
                    go to endchekch end;
               end;
          p:=0;
endchekch:
     end;
```

Figure A5-15. A new chekch

Exercises A5-6

1 - 4. For the flow charts prepared in Exercises 5-6, Main Text, write ALGOL procedures and/or calling programs.

Chapter A7

SOME MATHEMATICAL APPLICATIONS

A7-1 Root of an equation by bisection

Since finding the root of an equation is such a common problem, we shall write the ALGOL program in the form of a procedure called bisect. This program will correspond to the flow chart of Figure 7-5; then we will write an ALGOL program which calls this procedure:

> The parameters of the procedure are clearly a and b, the ends of the interval under consideration, a tolerance, epsi, which specifies the accuracy of the result, and a variable, root, which will be set equal to the root of the equation.

Figure A7-1 shows one way to code the bisect procedure. The first thing that is done in the body of bisect is to reassign the values of parameters a and b to auxiliary variables x1 and x2, respectively, as a safeguard to protect the values of arguments that match a and b in the calling program. We protect against the possibility of there not being a root in the interval by printing the message indicated in this case.

```
procedure bisect (a, b, epsi, root);
     real a, b, epsi, root;
begin
     real x1, x2, xm, temp, y1;
     x1 := a;
     x2 := b;
     y1 := f(x1);
     temp := y1 × f(x2);
     if temp > 0 then write ("Method is inapplicable")
     else if temp = 0 then
          begin if y1 = 0 then root := x1 else root := x2;
               write (a, b, epsi, root);
          end
     else begin
BOX6:          xm := (x1 + x2)/2;
               if abs(x2 - x1) ≥ ε then
                    begin temp := y1 × f(xm);
                         if temp < 0 then
                              begin x2 := xm; go to BOX6; end
                         else if temp > 0 then
                              begin x1 := xm; go to BOX6; end;
                    end
               else begin root := xm;
                         write (a, b, epsi, root);
                    end;
          end;
end  bisect
```

Figure A7-1

Now suppose we wanted to use this procedure to find the root of the equation $3x^3 - 7x + 2 = 0$ which lies between 1 and 2. Then we should need to define a real procedure which calculates the value of $3x^3 - 7x - 2$. If we choose epsi = 10^{-4} we would then need to call bisect by the statement

bisect(1,2,10^{-4},root).

The ALGOL program could be written as follows:

```
begin
    comment  Place the declaration of the procedure bisect here;
    real procedure f(x); real x;
        f:=(3 × x × x × x - 7) × x - 2 ;
    real root;
    bisect(1,2,10^-4,root);
end
```

Suppose we wish to use the bisect procedure with a series of functions. Though desirable, this would be difficult with bisect as written. We would have to reproduce the procedure bisect and "package" it with each function separately. What we need is some means by which to identify each of a series of functions so that we can communicate to the procedure bisect which of the functions it is to use when called. Most ALGOL processors take care of this problem by allowing the function itself to be a parameter of the procedure. Find out if your processor will allow you to do this. In this book we are assuming this is the case.

We now revise our procedure, renaming it "zero" and adding the function name f as a parameter. We also add the statement label L as a parameter (also permitted in most ALGOL processors). Now we can have a parameter list that is identical with the one in the funnel of Figure 7-6. The revised ALGOL procedure is given in Figure A7-2. It matches in nearly every respect the flow chart procedure given in the main text.

```
procedure zero (f, L, a, b, epsi, root);
     real a, b, epsi, root;
     real procedure f;
     label L;
begin
     real x1, x2, xm, temp, y1;
          x1 := a; x2 := b;
          y1 := f(x1); temp := y1 × f(x2);
          if temp > 0 then go to L
          else if temp = 0 then
               begin if y1 = 0 then root := x1 else root := x2; end;
          else begin
BOX6:          xm := (x1 + x2)/2;
               if abs(x1 - x2) ≥ ϵ then
                    begin temp := y1 × f(xm);
                         if temp < 0 then
                              begin x2 := xm; go to BOX6; end
                         else if temp > 0 then
                              begin x1 := xm; go to BOX6; end;
                    end
               else root := xm;
               end;
end  zero
```

Figure A7-2

Along with the two new parameters f and L. we notice two new declarations in the head of the procedure:

```
real procedure f
```

and

```
label L;
```

These declarations are necessary to convey to the compiler the vital information that the identifier f is not a variable but a placeholder for a function name. Similarly, we must tell the compiler that the identifier L has the meaning of (and is a placeholder for) a label.

Now let us use this second procedure to find the root of $3x^3 - 7x - 2 = 0$ between 1 and 2, the root of $x^5 - 4x^4 + 7x^3 - x + 3 = 0$ between -1 and 0 and the root of $x = \cos(x)$ between 0 and 1. Each of these equations can be thought of as a function of x. We give each a distinctive name:

$$f(x) = 3x^3 - 7x - 2 = 0,$$

$$g(x) = x^5 - 4x^4 + 7x^3 - x + 3 = 0,$$

and $$t(x) = x - \cos(x).$$

If we choose epsi = 10^{-4}, we could then write the ALGOL program shown in Figure A7-3.

```
begin
    real procedure f(x); real x; f := (3 × x↑2-7) × x-2;
    real procedure g(x); real x; g := (((x-4) × x+7) × x↑2-1) × x+3;
    real procedure t(x); real x; t := x-cos(x);
    comment Place the procedure declaration for zero here;
    real root;
    zero(f,label,1,2,.0001,root);
    write(root);
    zero(g,label,-1,0,2,.0001,root);
    write(root);
    zero(t,label,0,1,.0001,root);
    write(root); go to BOXx;
    label: write("Method is inapplicable"); BOXx:
end
```

Figure A7-3

We now show that it is possible, if we wish, to complete the analogy between our ALGOL zero and the flow chart procedure. Up to now we have protected the incoming values for the third and fourth parameters by reassigning these to the auxiliary or local variables x1 and x2 inside the procedure. This achieves the intent of the "wavy lines", i.e., x1 , x2 used in the funnel which tells us to protect these values. ALGOL gives us an optional way to achieve this that is closer in spirit to the wavy lines. We see this done in Figure A7-4 where we repeat the head and the first few statements of the body of a final version of zero.

```
procedure zero (f,L,xl,x2,epsi,root);
    value xl,x2;
    real xl,x2,epsi,root;
    real procedure f;
    label L;
begin
    real xm,temp,yl;
    yl := f(xl); temp := yl × f(x2);
    .
    .   Same as Figure A7-2
    .
end
```

Figure A7-4. Final version

Notice that we have used the parameter names xl and x2 in place of a and b, but have declared these in a special declaration

value xl,x2;

to be parameters whose values are being supplied to the procedure (i.e., slips of paper and not window boxes). Making xl and x2 each a "value parameter" means we no longer need the explicit protection mechanism we used before of writing assignment statements that transfer the values to local variables. If you decide to use the value declaration in any of the procedures you write, remember this simple restriction: A value declaration must be the first declaration in the procedure heading after the procedure name and the parameter list are given.

Exercises A7-1

1. Write an ALGOL "driver" program to solve all of the equations given in Exercises 7-1, Set C, main text. Use the indicated intervals and the indicated error tolerances. Include equation 3 a second time with epsi = 10^{-4}. Run the program and compare your results with the hand-calculated ones.

2. Write an ALGOL program to carry out the function evaluations needed in drawing the graphs in Exercises 7-1, Set A, main text. Run your program.

3. Write and run an ALGOL program to solve the alley problem (No. 6) in Exercises 7-1, Set D, main text. Then solve the problem to the nearest hundredth of a foot if the ladders are 25.83 and 19.14 ft. long and the crossover point is 7.17 ft. above the ground.

4 - 5. For each of the flow chart solutions you prepared for Problems 1 and 4, Exercises 7-1, Set D, employ the zero procedure (Figures A7-1 or A7-2) and write the companion ALGOL program and function procedures.

A7-2 The area under a curve: an example, $y = 1/x$, between $x = 1$ and $x = 2$

Since the area under the curve $y = 1/x$ is of interest in defining logarithms we begin by writing a simple ALGOL program for the calculation of the approximate area under this curve between $x = 1$ and $x = 2$. This calculation will provide an approximation to $\ell n 2$. We assume that an error tolerance epsi is read in from a card and that calculation of the approximate area is to be carried out by doubling the number of subdivisions each time and terminating the calculation when the absolute value of the difference of two successive approximations is less than epsi. The following program in Figure A7-5 follows closely the flow chart of Figure 7-16. Remember that $f(x) = 1/x$.

```
begin
    integer n, k;
    real epsi, AREA, S;
    array T[0:100];
    read (epsi);
    T[0]:=0.5 × (f(1) + f(2));
    n:=1;
BOX3: S:=0;
    for k:=1 step 2 until 2 ↑ n - 1 do
        S:=S + f(1 + k/2 ↑ n);
    T[n]:=0.5 × T[n-1] + S/(2 ↑ n);
    if abs(T[n] - T[n-1]) < epsi then go to BOX9;
    n:=n + 1;
    go to BOX3;
BOX9: AREA := T[n];
    write("AREA =", AREA);
end
```

Figure A7-5

Exercises A7-2

1. In the above program it is implicitly assumed that the calculation will terminate before n exceeds 100.

   (a) Is it possible for n to exceed 100 ?

   (b) What would happen if it failed to terminate before n exceeds 100 ?

   (c) Add some statements to the above program to protect against this undesirable event, even if the error tolerance is not satisfied.

   Print out a message in this case indicating failure to satisfy the error tolerance.

2. Criticize the above program for inefficiency. Revise it to make it more efficient by following the flow chart of Figure 7-17. Also incorporate a safety termination if n exceeds 100. Run your revised program using first 0.01 and then 0.001 as values for epsi.

3. Instead of terminating the calculation of the approximate area when the absolute value of the difference of two successive approximations is less than epsi, we could terminate the calculation after a fixed finite number of approximations have been calculated. Revise the program of the previous problem to read in an upper limit for the number of iterations to be carried out and then to terminate when this is reached. Run your program for n = 15.

4. Explain how to revise the program given in this section so that the calculation could be repeated for a series of values of epsi each of which is read in from a card.

5. Write an ALGOL program for the calculation described in Exercise 7-2, Set C, Problem 6, main text. Use f(x) = 1/x. Run your program and compare results using n = 5, 25, 75, 125, 200.

A7-3 Area under curve: the general case

We now consider the general case of finding an approximation to the area under a curve $y = f(x)$, above the x-axis and between the vertical lines $x = a$ and $x = b$. In order to make the program as useful as possible we shall write it in the form of a procedure. The function $f(x)$ is assumed to be defined as a real procedure. An error tolerance epsi is given and we terminate the calculation when the absolute value of the difference of two successive approximations is less than epsi. We follow the flow chart of Figure 7-20.

```
real procedure area(a,b,epsi,f); real a,b,epsi; real procedure f;
begin
     integer k,m;
     real h, S, OLAREA, NUAREA;
     m := 1;
     h := b - a;
     OLAREA := 0.5 × h × (f(a) + f(b));
BOX3: m := 2 × m;
     h := h/2;
     S := 0;
     for k := 1 step 2 until m-1 do
          S := S + f(a + k × h);
     NUAREA := 0.5 × OLAREA + h × S;
     if abs(NUAREA - OLAREA) < epsi then go to BOX9;
     OLAREA := NUAREA;
     go to BOX3;
BOX9: area := NUAREA;
 end area
```

If we want to use this procedure to calculate and print the approximate area under the curve $y = 1/x$, above the x-axis, and between the lines $x = 1$ and $x = 2$, we might use a tolerance of epsi = 10^{-4} and then we could write the following program:

```
begin
     real procedure f(x); real x; f := 1/x;
     comment Place the above procedure declaration for area here;
     real z;
     z := area(1,2,.0001,f);
     write(1,2,z);
end
```

Exercises A7-3

1. (a) Write an ALGOL function procedure area2(a,b,n,f) which calculates an approximation to the area under the curve $y = f(x)$, above the x-axis and between the lines $x = a$ and $x = b$ and which uses a subdivision of the interval (a,b) into n equal parts. Follow the flow chart drawn in Exercises 7-3, Problem 1, of the main text. Test your program for $y = \sin x$ between $x = 0$ and $x = \pi$ with $n = 5000$. (How does your result compare with the area of a semi-circle of diameter π ?)

   (b) Use function area2 to print out a table of natural logarithms for numbers from 1 through 51 in intervals of 5. Also, print out the library function ln(x) for the same values of x for comparison.

2. Tell how the procedure area (a,b,epsi,f) of this section may be adapted to protect against the possibility of an endless loop by causing termination of the calculation if the number of subdivisions exceeds n. If the calculation is terminated in this manner without satisfying the accuracy criterion, a message should be printed in addition to giving the approximation to the area.

3. Write an ALGOL program in which you first declare the procedure area of this section and the procedure area2 of Problem 1. Then call these procedures with appropriate values of the parameters to calculate approximations to the areas described below. First use 1,2,4 equal subdivisions of the interval and then use an error tolerance of epsi = 10^{-3}. Of course, you will need to supply the necessary real procedure declarations to define the functions which enter into the descriptions of the areas.

   (a) Below $y = .43429/x$, above x-axis, between $x = 1$ and $x = 3$. (The area is log 3.)

   (b) Below $y = 3x^2 + 2x + 1$, above x-axis, between $x = -2$ and $x = 2$.

   (c) Below $y = x^3$, above $y = x^2$, between $x = 1$ and $x = 4$.

4. Write an ALGOL program and necessary real procedures that can be used in calling on area(a,b,epsi,f) to compute an approximate value of π to four decimal places. (See Problem 6, Exercises 7-3, main text.)

A7-4 Simultaneous linear equations: Developing a systematic method of solution

In this section of the main text we explained carefully how to solve systems of two and three simultaneous equations. Exercises 7-4 provided examples of the method. You should now be ready to write a simple ALGOL program for the solution of two simultaneous equations in two unknowns.

Exercises A7-4

1. Follow the flow chart drawn in Exercises 7-4, Set B, of the main text, and write a corresponding ALGOL program for the solution of two simultaneous equations in two unknowns:

$$a_{11}x_1 + a_{12}x_2 = b_1$$

$$a_{21}x_1 + a_{22}x_2 = b_2$$

2. Use the program of Problem 1 to solve the following systems of equations on the computer. Make a hand-calculated check of your computer results. For systems (f) and (g), slide rule accuracy is sufficient.

(a) $4x - 2y = 5$
$2x + y = 4$

(b) $4x + 3y = 5$
$2x - 4y = 7$

(c) $3x - 4y = 12$
$4x + 6y = 3$

(d) $2x + 4y = -7$
$3x + y = 2$

(e) $5x + y = 2$
$3x - 4y = 7$

(f) $3.124x_1 + 5.375x_2 = -1.234$
$10.245x_1 - 5.214x_2 = 3.714$

(g) $5.128x_1 - 3.874x_2 = 12.42$
$3.817x_1 + 15.157x_2 = 3.784$

A7-5 Simultaneous linear equations: Gauss algorithm

In describing the solution of three equations in three unknowns we described each of the essential operations in turn and drew a flow chart for each. It will be instructive to build up the ALGOL program in the same gradual fashion. We begin by dividing the first equation through by a_{11}, as described in Figure 7-24. The corresponding ALGOL statements would be:

```
for  j:=2 step 1 until 3 do
     a[1,j]:=a[1,j]/a[1,1];
b[1]:=b[1]/a[1,1];
```

The elimination of x_1 from the i^{th} row, $i = 2,3$ is described in Figure 7-25 and the corresponding ALGOL statements would be:

```
for  j:=2 step 1 until 3 do
     a[i,j]:=a[i,j] - a[i,1] × a[1,j];
b[i]:=b[i] - a[i,1] × b[1];
```

Next we have to divide the new second equation by a_{22} and then eliminate x_2 from the third equation. Following these simple examples, you should have little trouble writing the ALGOL that's equivalent to Figures 7-27 through 7-30.

Exercises A7-5 Set A

1. Write the ALGOL statements corresponding to the flow chart of Figure 7-27.
2. Note the similarity between the statements of Problem 1 and those corresponding to Figure 7-24. Write a single set of ALGOL statements to cover both cases by following Figure 7-28.
3. Write the ALGOL statements for the flow chart of Figure 7-30.
4. Now write the ALGOL that's equivalent to Figure 7-33.

Next we want to carry out the back solution in order to obtain x_3, x_2, x_1 in turn. This is described in the flow charts of Figures 7-34 and 7-35. The ALGOL statements corresponding to the latter flow chart, Figure 7-35, would be:

```
for i := 3 step -1 until 1 do
begin
    x[i] := b[i];
    for j := 3 step -1 until i + 1 do
        x[i] := x[i] - a[i,j] × x[j];
end;
```

Now just as the complete flow chart of Figure 7-33 was built up from partial flow charts, so we can build up the complete ALGOL program corresponding to Figure 7-36 from the partial ALGOL programs which we have just discussed and which you have written in Exercises A7-5, Set A.

Exercises A7-5 Set B

1. Write a complete ALGOL program for the Gauss Algorithm given in Figure 7-36.

2. Run the above program on your machine and use the program to solve the systems of simultaneous linear equations represented by the following arrays:

   (a) $3x + 4y + z = -7$
   $2x + 4y + z = 3$
   $3x - 5y + 3z = 7$

   (b) $x + 2y - z = 4$
   $3x - 2y + 4z = 1$
   $x - 3y - 2z = 7$

   (c) $4x - 2y - 3z = 7$
   $3x - 5y + 2z = 1$
   $2x + y + 2z = 1$

   (d) $2x - y + 6z = 3$
   $3x - 4y + 4z = 1$
   $x + 2y - 5z = 7$

3. Now use the above program on your machine to solve these systems of equations:

   (a) $3.147x_1 + 2.419x_2 - 3.479x_3 = 4.219$
   $6.241x_1 - 5.678x_2 + 4.271x_3 = -52.17$
   $3.841x_1 + 5.761x_2 + 34.314x_3 = 27.14$

   (b) $27.147x_1 - 3.417x_2 - 3.479x_3 = 5.617$
   $31.468x_1 + 3.428x_2 + 4.719x_3 = 31.421$
   $11.121x_1 - 3.171x_2 + 5.314x_3 = -17.121$

Solution of n equations in n unknowns

The generalization to n equations is quite easy if we follow exactly the pattern we just used for 3 equations. You are asked to make the necessary changes in the partial programs in the following exercises.

Exercise F7-2 Set C

Revise your ALGOL program for the Gauss Algorithm to handle n equations and n unknowns, according to the procedure flow chart you prepared for Problem 2, Exercises 7-5, Set A, main text. Call the procedure Gauss. Test the procedure using the 4 by 4 system given in Problem 3 of Exercises 7-5, Set B, in the main text. Show the calling program which calls on Gauss.

*Exercise A7-2 Set D

1. In Exercises 7-5, Set C, of the main text you were asked to insert "partial pivoting" as a capability of your flow chart for the Gauss procedure. Show the corresponding changes necessary to the ALGOL procedure Gauss which you prepared in the preceding exercise.

2. Use the revised Gauss procedure to solve the following systems of simultaneous linear equations with and without partial pivoting.

(a) $3x_2 - 4x_3 = -4$
$3x_1 - 2x_2 + 4x_3 = 7$
$5x_1 + 15x_2 - 3x_3 = -4$

(b) $2x_1 - 3x_2 + 4x_3 = 7$
$4x_1 - 6x_2 + 13x_3 = 11$
$2x_1 - 7x_2 - 12x_3 = 1$

INDEX

ALGOL character set, 8
Algol 60, 1
alphanumeric data, 39
alternate exits from procedures, 109
area under a curve
  from x = 1 to x = 2, 126
  general case, 128
arithmetic expression, 22
array declarations, 74
  for doubly-subscripted variables, 78
array input and output, 75
array storage, 73
assignment meaning when there are type differences, 32
assignment statements, 22

bisection process, 119
block, 97
body of a program, 36
branching, 45
  from procedures, 109
Burroughs Algol write statement, 56

calling a procedure, 107
card layout, 6
composition of function designators, 105
compound conditions, 62
compound statement, 36
computer program, 2
conditional statement, 45
  else type, 49

double subscripts, 78
dummy statement, 61

for clause, 81
for statement, 81
format, 57
function designator, 100
function names as procedure parameters, 110
function procedure, 98, 103
function reference, 28
functions as procedure parameters, 122

Gauss algorithm, 131
go to statement, 4
greatest integer function, 25

HALT, 61
head of a program, 36

identifiers, 10
if statement, 45
identifying remarks in output, 55
input-output statements, 15
integer
  division, 25
  procedure, 98
  type variable, 10
iteration, 81

labels, 10
  as procedure parameters, 110, 122
local variable, 104
looping, 81

multiple branching, 62

nested conditionals (2-way branches only), 58
nested loops, 93
nested procedure declarations, 116
non-local variable, 104
numerical constants, 8

operator symbols, 13
order of computation, 31

parameter list of a procedure, 104
procedure
  bisect (a, b, epsi, root), 120
  body, 97
  call, 107
  declaration, 97
  head, 97
  zero (f, L, a, b, epsi, root), 122
procedures, 97
program, 37
proper procedures, 106

read statement, 16
real numbers, 7
real procedure, 98
real type variable, 10

simple statement, 36
simultaneous linear equations, 130
source programs, 2
spaces, 14
special symbols, 13
standard mathematical functions, 11
STOP, 61
storage of doubly-subscripted arrays, 78
string, 112
string procedures, 112
subscripted variables, 73
symbol manipulation, 111
  in extended ALGOL, 111

table-look-up, 91
target program, 2
type of an evaluated expression, 23

unary minus, 30

value declaration, 124
variable (value) protection in ALGOL, 123
variables, 10

write statement, 19